Szemantikus adatok lekérdezése federált és osztott rendszereken

Doktori értekezés

Gombos Gergő

Témavezető: Dr. Kiss Attila

Eötvös Loránd Tudományegyetem
Informatikai Kar
Információs Rendszerek Tanszék

Informatikai Doktori Iskola
Iskolavezető: Prof. Csuhaj-Varjú Erzsébet
Doktori Program: Információs Rendszerek
Programvezető: Prof. Benczúr András

Budapest, 2018

Köszönetnyilvánítás

Szeretnék köszönetet mondani Kiss Attila témavezetőmnek, aki segítséget nyújtott a cikkek elkészülésében, valamint minden lehetséges módon támogatta a kutatásokat.

Köszönetet mondok Benczúr Andrásnak, aki cikkek megjelenésében támogatott.

Köszönetet mondok a doktorandusz kollégáknak, Matuszka Tamásnak, Rácz Gábornak, Pinczel Balázsnak és Szücs Imrének, akikkel közös publikációk készültek és akiktől sokszor hasznos kritikákat kaptam.

Köszönetet mondanék még a tanszéki dolgozók közül Laki Sándornak, akitől rengeteget tanultam a kutatói életről és hasznos tanácsokkal látott el a kutatások során, illetve Vincellér Zoltánnak, akivel több projektben is részt vettem, és nem utolsó sorban Frankó Gáborné Mariannak a rengeteg adminisztratív segítségért.

Köszönetet szeretnék mondani Vattay Gábornak, aki projektek segítségével támogatta a kutatásaimat.

Köszönöm még Stéger Józsefnek a hasznos tanácsokat és a közös munkákat.

Köszönettel tartozom szüleimnek, testvéreimnek hogy segítettek és támogattak az egyetemi évek alatt.

Végül és nem utolsósorban köszönöm Dióssy Emesének a dolgozathoz nyújtott segítségét.

A dolgozat elkészültét és a kutatásokat a következő projektek támogatták: EFOP-3.6.3-VEKOP-16-2017-00002, FP7 NOVI (Network Innovation over Virtualized Infrastructures), FP7 FI-PPP (XiFi federált infrastruktúra fejlesztése), FP7-2013-ICT-FI FI-Core, FuturICT.hu (grant no.:TAMOP-4.2.2.C-11/1/KONV-2012-0013), TÁMOP 4.2.1/B-09/1/KMR-2010-0003, CNL (Communication Networks Laboratory, Ericsson-ELTE).

„A repülésnek is megvan a maga művészete, vagyis inkább fortélya. Abban rejlik, hogy megtanuld magad a földre vetni és elhibázni azt. Válassz ki egy derűs napot (...), és próbálgasd. Az első rész könnyű. Mindössze annyi kell hozzá, hogy képes légy teljes súlyoddal a földre vetődni, azzal az elhatározással, hogy nem baj, ha fájni fog. Ugyanis ha nem sikerül elhibázni a földet, akkor fog.„

- Douglas Adams, Galaxis útikalauz stopposoknak

„Miért mászik meg az ember egy hegyet? - Mert ott van!„

- George Mallory, hegymászó

Osztott rendszerek definíciója: *„Akkor tudhatod, hogy ilyennel dolgozol, ha egy általad soha nem hallott számítógép meghibásodása megakadályoz téged bármiféle munka elvégzésében.„*

- Leslie Lamport

Tartalomjegyzék

Ábrák jegyzéke

Táblázatok jegyzéke

Algoritmusok és példák jegyzéke

1. fejezet

Bevezetés

Az információk tárolása mindig is fontos feladata volt az emberiségnek. Eleinte ezeket papírra vetettük, majd a számítógépek megjelenésével különböző háttértárolókon tároltuk. Az Internet térhódításával az információk már nem csak lokálisan érhetőek el, hanem a világ bármely pontjáról. A mindennapjaink részévé vált az információk Interneten keresztüli megszerzése. Ahhoz, hogy megtaláljunk egy bizonyos információt a világhálón a Google kifejlesztett egy hatékony indexelő algoritmust a weboldalakon található szavakra. Amikor a keresőben megadunk szavakat, akkor azokat az oldalakat fogja nekünk visszaadni eredménynek, amelyeken ezek a szavak szerepelnek. Ez nem minden esetben lesz helyes, hisz ha olyan szavakra keresünk rá, amelyeknek több értelme is van, akkor olyan találatokat is kapunk, amelyek számunkra nem fontosak. Vegyük például a 'nap' szót. Ekkor megkapjuk a nap mint égitest wikipédia oldalát, de kapunk olyan oldalakat, ahol a hét napjáról van szó. Ha viszont mi az 'egy nap' című filmre akarunk rákeresni, akkor ez a szó alapú keresés kevésnek bizonyul.
Az Internet atyja Tim Berners-Lee emiatt egy újfajta Internetet vázolt fel a 2000-es években, amit szemantikus webnek [26] hívott. Ennek lényege, hogy ne csak dokumentumokat tároljunk a világhálón, hanem a dokumentumokban tárolt információkat is. Ehhez egy új tárolási módra van szükség, amely egységes és amelyhez nincs szükség az Internet teljes megváltoztatására. Ha vesszük az előző példánkat és azt írjuk be a keresőbe, hogy 'nap film', akkor már nem csak olyan oldalakat fogunk megtalálni amelyen szerepelnek ezek a szavak, hanem olyanokat is, amelyek az 'egy nap' című filmhez tartoznak, mint például a film előzetese. Ahhoz, hogy a Google keresője így tudjon működni, értelmezni kell a weboldalakon található információkat. Ha tudja, hogy az adott oldal a filmről szól, akkor a kereső ezt is meg fogja találni nekünk eredményként. A háttérben természetesen a szemantikus web alapjai dolgoznak és az oldalak indexelése már nem csak szavak, hanem egy tudásbázis alapján történik.

A probléma ezzel az, hogy ezek a tudásbázisok nagy és strukturálatlan adatokat tárolnak, amelyek kezelése kihívást jelent. A dolgozat ezeknek a tudásbázisoknak a lekérdezésével foglalkozik. A probléma megértéséhez meg kell értenünk a szemantikus web működését.
A weboldalakon tárolt tudást olyan formában tároljuk, hogy az mindenkinek egyértelmű legyen. Emiatt minden objektumot, fogalmat egy egyedi azonosítóval kell ellátni. Ennek az azonosítónak olyannak kell lennie, hogy illeszkedni tudjon a világháló működésébe. Emiatt az azonosítókra az Internetes URL-ket használjuk. Ezeket IRI-nek (Internationalized Resource Identifier) [30] vagy URI-nak (Uniform Resource Identifier) [36] is szokták nevezni. Ezeket az azonosítókat különböző konzorciumok, kutatók vagy szakmai csoportok alkotják meg, de bárki létrehozhat saját azonosítókat. Ennek vannak előnyei és hátrányai is. Ha van egy azonosító, amit ismerünk, akkor minden információ összegyűjthető arról az objektumról vagy fogalomról. Ha nincs még ilyen azonosító a szemantikus weben, akkor létre tudunk hozni egyet. A hátránya viszont akkor jön elő, ha egy meglévő elemnek hozunk létre egy új azonosítót. Ilyenkor ugyanaz a fogalom többször fog szerepelni és nem minden információ lesz összekapcsolható az azonosítókkal. Az ilyen azonosítók keresése egy probléma terület a szemantikus web világában, amellyel most nem foglalkozunk.
Az azonosítókkal még nem tudunk információt tárolni. Az információ tárolását az RDF (Resource Description Framework) [41] szabvány írja le. Az információ tárolásának ötlete, hogy az információkat egyszerű állításokként tároljuk. Minden állítás egy alanyból, egy állítmányból és egy tárgyból fog állni. Ha az előző példánkat folytatjuk, akkor az 'egy nap' című filmről azt tudjuk mondani, hogy '*az egy nap típusa film*'. Ekkor három IRI-re van szükségünk: egy magát a filmet fogja azonosítani, egy a filmet, mint fogalmat és egy a típust, mint fogalmat. Az szabvány rendelkezik alap szókinccsel, amely tartalmazza a *típus*-t mint fogalmat, így mindenki ugyanazt az IRI-t fogja használni, ha valaminek a típusát akarja leírni. Az állításokat *hármasok*-nak fogjuk nevezni, és egy adott azonosítóhoz több hármas is tartozni fog. Az 'egy nap' című filmhez lesz állítás a rendezőjéről, a bemutatójának évéről vagy a kategória besorolásáról. Az így kialakult hármashalmaz lesz az, amit mi szemantikus adatoknak vagy tudásbázisnak nevezünk. Ez az adathalmaz az, amit fel fogunk használni arra, hogy információkat nyerjünk ki belőlük.
Az információ kinyeréséhez egy lekérdezőnyelvre van szükségünk, ami a szemantikus webnél a SPARQL [42], melynek nyelvezete hasonló az SQL-hez. A lekérdezőnyelv többféle lekérdezési típussal is rendelkezik, amit a dolgozat későbbi részében be is mutatok. Jelen esetben csak lekérdezni szeretnénk az adatokat, így csak a SELECT típusú lekérdezést mutatom be. A lekérdezés három fő részből tevődik össze. Van egy SELECT rész, ahol

megadjuk azokat a változókat, amelyeket meg szeretnénk kapni, mint eredmény. Van egy WHERE rész, ahol megfogalmazzuk, hogy az egyes változókra milyen feltételeknek kell teljesülniük. Végül egy eredmény módosító rész, ahol például rendezni (ORDER BY) vagy épp limitálni (LIMIT) tudunk. A különbség az SQL-hez képest, hogy a feltételeknél úgynevezett *hármasmintákat* fogalmazunk meg. Ezek a *hármasminták* olyan hármasok, ahol valamely pozícióban (alany, állítmány, tárgy) egy változó szerepel. Egy változó több hármasmintában is szerepelhet és csak azok a változó által felvett értékek lehetnek eredmények, ahol minden feltétel teljesül rájuk.
Az eddig bemutatott lekérdezés akkor működik, ha csak egy adathalmazunk van. A szemantikus web és a Linked Open Data (LOD) [66] céljai között szerepel, hogy az információknak összekapcsolhatóknak kell lenniük. Ez úgy történhet, hogy egy IRI megjelenhet több adathalmazban is. A lekérdezések futtatásánál viszont felmerülnek kérdések: Ki végezze el a kiértékelést? Milyen sorrendbe végezzük az egyes lekérdezéseket? Hogyan kapcsoljuk össze a részeredményeket? Ezzel a problémával foglalkozik az úgynevezett *federált lekérdezések* [111] terület.
A dolgozatomban a szemantikus web használhatóságára mutatok be módszereket, technikákat. Bemutatom, hogyan lehet használni a *federált rendszereket* vékonykliensek kiszolgálására, illetve hogyan alkalmazhatóak a szemantikus webet nem ismerő felhasználók segítésére, hisz egy SPARQL lekérdezés megfogalmazása igényel némi tudást. Végül a szemantikus adathalmazok nagy méretéből és komplexitásából adódó problémákkal foglalkozok, ahol azt mutatom meg, hogyan lehet a Big Data eszközöket használni arra, hogy megválaszoljanak szemantikus lekérdezéseket.
A dolgozat fő iránya az elosztott rendszerek, hisz a kliens-szerver architektúra az alapjai az osztott rendszereknek, és az osztott rendszerek alapjai a Big Data eszközöknek, ahogy azt Tanenbaumnál [55] is olvashattuk. A következő fejezetekben részletesebben bemutatom az egyes témaköröket.

1.1. Szemantikus adatok kliens-szerver architektúrán

A szemantikus web gondolata egy jó irány az információk tárolására, de ugyanez a tárolás több problémát is felvet. Első problémaként felvethetjük, hogy a lekérdezések futtatásának nagy az erőforrásigénye. Egy szemantikus adatbázisban keresni vagy a világhálón található adathalmazokban lévő információkat összekapcsolni költséges művelet. Felmerül az a kérdés, hogy a vékonykliensek hogyan tudják ezt megoldani. Egy adatbázis-kezelő esetében a kliens egy lekérdezést küld az adatbázisszervernek, amely optimalizálja és kiér-

tékeli azt. A szemantikus web esetében ez azért problémás, mert egyfelől több adatforrásunk is lehet különböző szerveren, vagyis nincs központosítva az adatok tárolása. Másfelől a tárolt adatok strukturálatlanok, amiről tudjuk, hogy bennük történő keresés lassú. A mobil eszközök kifejezetten vékonykliensek, ahol a számítások a készülék üzemidejét csökkentik. Erre jelentenek megoldást a kliens-szerver architektúrák, amellyel a dolgozat 3. fejezetében foglalkozom. Egy kliens-szerver architektúrán egy erősebb szerver végzi a költséges számításokat és csak az eredményt adja vissza a klienseknek. A szemantikus adatokat tipikusan egy *SPARQL végponton* keresztül tudjuk elérni. Ezek a végpontok, olyan API-k, amelyek szabványos kéréseket képesek fogadni és a választ a megfelelő formátumban adják vissza. Több publikáció [40, 56] jelent meg olyan alkalmazással, amely egy szemantikus adatokat kezelő szervert alkalmaz. Ezek az alkalmazások tipikusan egy dedikált végponthoz csatlakoztak, amely elvégezte a számításokat. Ha viszont több adatforrásból akarjuk az információkat összekapcsolni, akkor csak azok a szerverek képesek a lekérdezést megválaszolni, amelyek képesek a SPARQL 1.1-es [80] verzióját futtatni. Ekkor olyan lekérdezésekre van szükség, amelyben megadjuk, hogy az egyes hármasminták mely adathalmazra vonatoznak. A SPARQL 1.0-ás szabványban ilyen lehetőségünk még nem volt, de a SPARQL 1.1 megjelenésével használhatjuk a SERVICE kulcsszót. Egy lekérdezés WHERE feltételén belül több SERVICE részt is alkalmazhatunk. Minden résznek meg kell adni a SPARQL végpont elérésének URL-jét, majd meg kell adnunk azokat a hármasmintákat, amelyek arra a végpontra vonatkoznak. Ekkor a SPARQL végpont megfuttatja az egyes hármasminta halmazokat az adott végponton, majd az eredményeket összekapcsolva visszaadja a végső eredményt. Ezeket a lekérdezéseket eléggé korlátozzák a SPARQL végpontok, hisz egy részeredmény elég nagy is lehet. A nagy részeredmény fogadása hálózati terhelést jelent a szervernek, ami korlátozza a szerver rendelkezésre állását. Valamint az eredmények tárolása memóriát emészt fel, ami akár a szerver összeomlásához is vezethet. Ha ezt a feladatot egy kliens szeretné elvégezni, akkor megint csak abba a problémába esünk bele, hogy a kliensnek elfogy az erőforrása.

A felhasználó szempontjából viszont érezhetjük, hogy egy ilyen lekérdezés megfogalmazása is elég nehézkes, hisz ismernünk kell a végpontok elérését és az ott tárolt adatokat, valamint az összekapcsolási lehetőségeket. Emiatt indult el egy olyan kutatási irány, amely azzal foglalkozik, hogy hogyan lehet a SPARQL 1.0-as lekérdezéseket (ezekben nem szerepel a SERVICE kulcsszó) lefuttatni több végponton. Az erre képes rendszereket hívják *federált rendszerek*nek. A dolgozat több szempontból is vizsgálja ezt a problémát. A 3.4. fejezet, olyan kliens-szerver megoldást mutat be, ahol a kliens egy mobil eszköz, a szerver pedig egy federált rendszer. Az ilyen megoldások csökkentik a mobil terheltségét és lehe-

tővé teszik a mobil eszközök számára a szemantikus web használatát. A 4. fejezet pedig azzal foglalkozik, hogyan lehet hatékonyabbá tenni a federált rendszerek [108] működését. Ehhez meg kell néznünk hogyan is működik egy federált rendszer. Erről szól a következő fejezet.

1.2. Federált rendszerek szemantikus adatokhoz

A federált rendszerek működése több részből tevődik össze. Bemenetként egy SPARQL 1.0-as lekérdezést kapnak, amelyek nem tartalmaznak SERVICE kulcsszót és nem tartalmaznak semmilyen információt arról, hogy melyik végponton kell lekérdezni őket. A lekérdezés csak *hármasmintákat* tartalmaz. A rendszer első feladata, hogy minden hármasmintához megtalálja azt a végpontot, amely választ tud adni. Ezután az információk alapján olyan al-lekérdezéseket készít amelyek egy végpontra vonatkoznak. A kapott részeredményeket ezután összekapcsolja és elkészíti a végső eredményt, amely a válasz lesz a lekérdezésre. A rendszer kritikus része a végpont választás, hisz ha olyan végponton futtatjuk le a lekérdezést, amely nem tartalmazza az adatokat, akkor üres eredményt kapunk vissza. A megfelelő végpont kiválasztásához valamilyen információra van szüksége a rendszernek. Erre kétféle megoldás ismert. Az első nem tárol információkat, hanem úgynevezett ASK lekérdezéseket futtat az egyes végpontokon, ami egy a SPARQL 1.0 négy lekérdezési fajtájából. Ennek a lekérdezésnek a felépítése egy WHERE részből áll, ahol a *hármasmintákat* adjuk meg. A visszatérési értéke egy igaz-hamis lesz aszerint, hogy a végponton található-e olyan adat, amely megfelel a feltételeknek. Ez azért hatékonyabb, mint egy SELECT lekérdezés, mert az ASK lekérdezés megáll az első találat után, így nem fut végig a teljes adathalmazon. Az ASK lekérdezést használó federált rendszer a kapott eredményekből tudja, hogy az egyes hármasmintára, mely végpont tudja megadni a választ és így készíti el az al-lekérdezést. A másik megoldás, hogy egy katalógust használunk arra, hogy az egyes végpontok milyen állításokat tartalmaznak. Azért az állítmányokat tárolja, mert egy lekérdezésben szereplő hármasminta legtöbbször tartalmazza ezt az információt, még az alany és a tárgy sokszor egy változó. A hármasmintákban található állítmányok alapján a rendszer elkészíti az al-lekérdezéseket, és lefuttatja a megfelelő végpontokon, végül a kapott eredményeket összekapcsolja a változók alapján. A dolgozat 4.4. fejezete a federált lekérdezésekkel foglalkozik. Azt vizsgálja, hogyan lehet használni az adathalmazban található IRI-k előtagját a végpontok azonosítására. Az előtagok a hosszú URL-k prefixe. Például a *http://dbpedia.org/resource/Garfield_(character)* IRI-nek a prefixe a *http://dbpedia.org/resource/*, amit röviden *dbr*-nek is szoktak leírni. Az

ötlet azon alapszik, hogy egy adathalmaz jellemzően egy előtagot használ az ott tárolt IRI-khez.

1.3. SPARQL ajánlórendszer

A következő szempont, amiben a szemantikus web használhatósága problémás, hogy hogyan tudják a felhasználók használni. Ha egy olyan embert kérnénk meg, hogy írjon egy SPARQL lekérdezést, aki nem ismeri a szemantikus webet, akkor nem lenne képes elkészíteni azt. Ennek oka, hogy az adatok nem egy adatbázisban vannak, ahol meg lehet nézni a táblákat, vagy a táblák szerkezetét és ez alapján elkészíteni a lekérdezést, hanem az adatok a világhálón több adatbázisban vannak tárolva, amelyekről vagy van leírás vagy nincs. Másik probléma, hogy ismernie kell a végpontok elérését. Ezek jellemzően nem egységesek, így ezeket is nehéz megtalálni. Több projekt foglalkozik azzal, hogy ezek a végpontok hogyan érhetőek el és hogy milyen a rendelkezésre állásuk [80, 118]. Manapság elterjedt módszer az automatikus kiegészítés funkció, melynek lényege, hogy miközben gépelünk a kliensen ajánlatokat kapunk az alapján, amit eddig írtunk. Ez a módszer működik a keresőknél, böngészőknél, de a mobil eszközöknél is. A háttérben a kliens lekérdezéseket futtat és az eredmény alapján készíti az ajánlásokat. Egy SQL kliens szintén rendelkezik ilyen funkcióval, ami a lehetséges táblaneveket vagy oszlopokat ajánlja fel nekünk. Ez a módszer szintén segítségére lehet azoknak a felhasználóknak, akik SPARQL lekérdezést szeretnének írni. Ennek a módszernek a bemutatásával foglalkozik a 4.5 fejezet. Az ötlet lényege, hogy egy federált rendszer kérdezze le a végpontokat az eddigi félkész SPARQL lekérdezés alapján. A válaszul kapott eredményekből pedig készítsen ajánlásokat, amelyből a felhasználó egyszerűen ki tudja választani a megfelelő hármasmintát. Az újonnan hozzávett hármasminták alapján újabb hármasmintákat ajánlhat a rendszer.
A módszer hasznos a felhasználó szempontjából, de más megközelítésből is alkalmazható, hisz miközben készül a lekérdezés, a federált rendszer információkat is kap az adott végpontról. Mivel a végpont kiválasztás kritikus része egy federált rendszernek, így hatékonyabbá tudjuk tenni, ha további információt adunk hozzá. A kiválasztás lényege, hogy azokat a végpontokat alkalmazzuk a kiértékelésnél, amelyek az adott hármasmintát ajánlották. Ez mind a két végpont választási stratégiánál működik, hisz az ASK lekérdezést használó megoldásnál kevesebb végpontot kell megvizsgálni, még a katalógus megoldásnál csökkenteni tudjuk a katalógus méretét. Ezekről az eredményekről részletesebben írok a 4.6 fejezetben.

1.4. SPARQL osztott kiértékelése Big Data eszközökkel

Ahogy azt már többször említettem a szemantikus adatok nagy és strukturálatlan adatok, emiatt tudnak a federált rendszerek jól működni, mert elosztva számolódnak ki az egyes lekérdezések részeredményei. A világ gyorsulásával a generált adatok mennyisége rohamosan nő, emiatt fontos témakör lett a nagy adatok elemzése, lekérdezése. Ezt nevezzük *Big Data*-nak. Rengeteg eszköz jelent meg, amely a nagy adatok elemzését teszi lehetővé. Emiatt egy új irány alakult ki, amely arra keres megoldásokat, hogy hogyan tudjuk alkalmazni a Big Data eszközöket szemantikus adatok lekérdezésére. Az egyik legismertebb rendszer a Hadoop [98], amely egy open-source megvalósítása a Google által kitalált MapReduce [58] paradigmának. Ez a rendszer a klaszterben futó gépeken tud hatékonyan működni. Az alapja az "oszd meg és uralkodj"-elv, vagyis a feladatot kisebb, úgynevezett Map folyamatokra osztjuk, amelyeknek az eredményeit a Reduce folyamatok fogják feldolgozni. Az adatok tárolása elosztva történik a klaszter gépein, az elosztott fájlrendszeren, a HDFS (Hadoop Distributed FileSystem)-n [77]. Az adat betöltéskor a Hadoop úgynevezett *chunk*-okra darabolja az adatokat, amelyek alapértelmezetten 128 MB méretűek. Ezután ezek a chunk részek tárolódnak a klaszter gépeire. A Map folyamat mindig azon a gépen próbál futni, ahol az adott file chunkjai megtalálhatóak. Az előnye ennek a rendszernek a jó hibatűrő képessége. Ha egy gép kiesik valamilyen hiba miatt, akkor is megtalálható lesz az adat valamely másik gépen a többszörös tárolás miatt. Ahogy korábban említettem egy erőforrásról több állítás is létezik. Az állítás tárgy részén szintén egy erőforrás lehet, és erre szintén mondhatunk állításokat. Ha máshogy képzeljük el ezt az adathalmazt, akkor ez nem más, mint egy gráf. Az alany és a tárgy egy-egy csúcs és a köztük lévő él az állítmány. Az élnek a címkéje maga az állítmány IRI-je. A szemantikus adatokban az *RDFS* [32] szókészlettel írhatunk le osztályok közötti kapcsolatot. Egy adott IRI osztály, ha az *rdf:type* állítmány tárgyában az *rdfs:Class* szerepel. Ezután ha alosztály kapcsolatot szeretnénk megfogalmazni két osztály között, akkor azt az *rdfs:subClass* állítmánnyal tehetjük meg. Ha meg akarjuk ismerni egy adathalmaz osztálystruktúráját, akkor vehetjük ezeket az állítmányokat, és csak ezeket hagyjuk meg az adathalmazban. Ennél érdekesebb, hogyha meg tudnánk azt is, hogy az egyes osztályokból hány elem található. A 5.3 fejezetben bemutatott megoldás a gráfot csökkenti úgy, hogy csak bizonyos éleket hagy meg és bizonyos csúcsokat von össze. Ezután ezeket a gráfokat már meg tudjuk jeleníteni egy gráfmegjelenítő szoftver segítségével.

A Hadoop rendszer után megjelentek újabb és újabb keretrendszerek, mint például az Impala, a Pig, az HBase, stb. Mindegyik különböző feladatra, különböző típusú adattárolásra, vagy adatelemzésre lett kifejlesztve. Az Impala kicsit kilóg a sorból, mert saját

deamon-okkal kapcsolódik a háttértárhoz és saját ütemezőt alkalmaz a folyamatok futtatásához. A Hadoopnak egyik problémája viszont, hogy az egyes számítási lépések között a háttértárat használja, amik lassabbak a memóriáknál. Ebből a megfontolásból készítették el a Berkley egyetemen a Spark keretrendszert, amely az adatot nem a merevlemezen tárolja elosztva, hanem a klaszterben lévő gépek memóriájában. Ehhez egy új típust vezettek be, ami az RDD (Resilient Distributed Datasets) [99]. Ez egy elosztott memóriaalapú objektum, amivel az elosztott számításokat gyorsabban lehetett elvégezni. A Spark megjelenésével újra felmerült a szemantikus adatok lekérdezési lehetősége Hadoop-alapú rendszereken. Ebből szintén több eredmény született, amelyeket a dolgozatban később részletezni fogok. A Spark nagy előnye, hogy több olyan keretrendszer is épült rá, amely képes elosztott környezetben problémás elemzéseket is elvégezni. Ilyen például a stream-alapú adatelemzés, gépi tanulás vagy a gráfelemzés. Ahogy már korábban említettem, a szemantikus adathalmazok lényegében gráfok. Felmerül a kérdés, lehet-e használni egy elosztott gráfelemző rendszert arra, hogy szemantikus lekérdezéseket válaszoljon meg. Az elosztott gráfelemzés a BSP (Bulk Synchronous Parallel) [22] modellen alapuló Pregel [76] modellt alkalmazza. Ennek a modellnek a lényege, hogy az iteratív feldolgozásban az egyes folyamatok külön-külön tudjanak párhuzamosan számolni, majd az eredményeiket egymás között megosztani. Az új kapott eredménnyel újra párhuzamosan tudnak számolni. Az osztott gráfelemzés hasonlóan működik. Itt a feldolgozóegységek a csúcsok lesznek. Az egyes csúcsok az eredményeiket csak az éleken keresztül küldhetik el a szomszédaiknak. A rendszer aggregálja ezeket az üzeneteket, majd odaadja a csúcsoknak. Minden csúcs egy üzenetet fog megkapni egy iterációban, és ez alapján számol tovább. A 5.4., 5.5. fejezetekben bemutatok olyan algoritmusokat, amelyek SPARQL lekérdezéseket tudnak kiértékelni a Spark segítségével.

1.5. A dolgozat felépítése

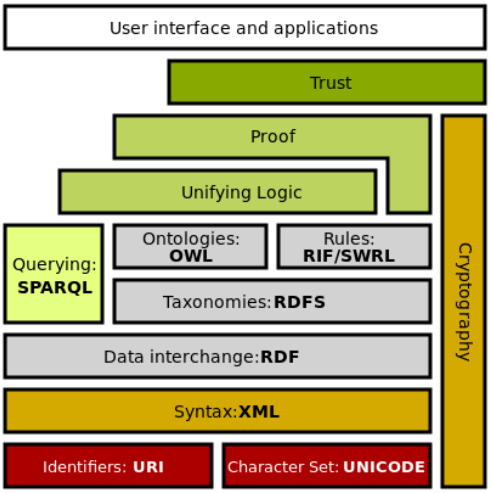
Először a formális és informális bemutatása történik a szükséges rendszereknek, algoritmusoknak és fogalmaknak a 2. fejezetben. Ezután a kliens-szerver architektúrára épülő rendszerek bemutatása olvasható a 3. fejezetben (Tézis 1.). Ezután a 4. fejezet bemutatja a federált rendszerek működését, ASM [93] segítségével egy formális modellt ad, valamint leírja a hármasminta ajánló rendszert (Tézis 2.) és annak használatát a végpont választásnál (Tézis 3.). Végül az 5. fejezet bemutatja a Big Data eszközök alkalmazását szemantikus adatok elemzésére (Tézis 4.) és lekérdezésére (Tézis 5., Tézis 6.). Végezetül a 6. fejezet összegzi az eredményeket és a téziseket.

2. fejezet

Elméleti áttekintés

2.1. Szemantikus web

A szemantikus web ötletét Tim Berners-Lee publikálta 2001-ben [26]. A fő elgondolása az volt, hogy az Interneten megtalálható információk olyan formában legyenek tárolva, amiket a gépek is tudnak értelmezni. Máshogy megfogalmazva, az Interneten lévő információkat egészítsük ki olyan leírásokkal, amelyek alapján további információt kaphatunk. A szemantikus web egy olyan infrastruktúra, ami több komponensből tevődik össze. Ezek meghatározásával több publikáció is foglalkozott [34, 35, 38, 60]. A 2.1. ábrán látható a szemantikus web ma alkalmazott felépítése. Legalul az adattárolás található. A szemantikus weben minden fogalomnak és objektumnak egy egyedi azonosítóra van szüksége. Ezt reprezentálja az URI [36] komponens. Ezeknek az azonosítóknak, olyannak kell lenniük, amelyek elérhetőek a világhálón, így ezek tipikusan URL címek, amelyek UNICODE kódolásúak. Ezután következik az adatok összekapcsolásának és az állításoknak a kifejező nyelve az XML. Ez egy ismert nyelv, amely használatos az Internet világában, emiatt alkalmazzuk a szemantikus webnél is. Az adatok leírásának szabályát az RDF (Resource Description Framework) [41] keretrendszer összegzi. Ez tartalmaz több adattárolási formátumot, illetve alapszókincset. Az RDFS [32] egy olyan szókészlet, amely az osztályok és a köztük lévő kapcsolatok megfogalmazásához tartalmaz kulcsszavakat. Ilyen lehet például az *rdfs:range*, amely azt adja meg egy állítmányra, hogy a tárgya milyen osztályú lehet. Ezután következik a SPARQL [44], ami a szemantikus web lekérdező nyelve. Részletesebb osztály leírás készíthető az OWL nyelv [65] segítségével, amely már komplexebb kapcsolatokat (tranzitivitás, szimmetria, stb.) tesz lehetővé az osztályok között. A szemantikus web fontos része a logikai következtetés, amelyhez a RIF (Rule Interchange Format) [62] és a SWRL (Semantic Web Rule Language) [31] leírónyelvek nyújtanak segítséget. Ezeket



2.1. ábra. Szemantikus web rétegei: ez a ma használatos architektúra a szemantikus webnél.[2]

a logikai leírásokat egy logikai következtető rendszer segítségével tárhatjuk fel, majd az így kapott eredményt a SPARQL segítségével kérdezhetjük le. A további komponensek csak tervezés szintjén léteznek. Ezek felelnek azért, hogy az adatok megbízhatóak, összekapcsolhatóak vagy éppen titkosítva legyenek. Végül a legfelső réteg pedig a felhasználói réteg.

A dolgozatban a SPARQL lekérdezésekkel és a felhasználói alkalmazásokkal foglalkozom.

2.1.1. RDF, Adatformátumok

Az RDF (Resource Describtion Framework) egy olyan keretrendszer, amit a szemantikus web adatainak és azok összekapcsolásának tárolására használunk. Az RDF lényegében egy címkézett irányított gráf formátum, ami azt jelenti, hogy minden csomópontnak és élnek van címkéje. Az első lépés, hogy minden objektumnak adjunk egy egyedi azonosítót. A 'Garfield' karakter azonosítója például a *http://dbpedia.org/resource/Garfield_(character)*. Ezután ezekből az azonosítókból állításokat készítünk, alany-állítmány-tárgy formában. A 1. példán láthatjuk ezeket az állításokat, amelyek leírják, ki készítette ezt a karaktert, mikor és hol jelent meg először, illetve, hogy hol született a szerző. Ezt a leírást nevezzük N-triples [113] formátumnak. Ez az egyik legegyszerűbb formátuma a szemantikus webnek. Az állításokat egymás alá írjuk és ponttal választjuk el őket. A legelterjedtebb formátum viszont nem ez, hanem az RDF/XML, amit a 2. példa mutat be. Ez a leírás volt az első formátuma a szemantikus

[2] `https://commons.wikimedia.org/wiki/File:Semantic_web_stack.svg`

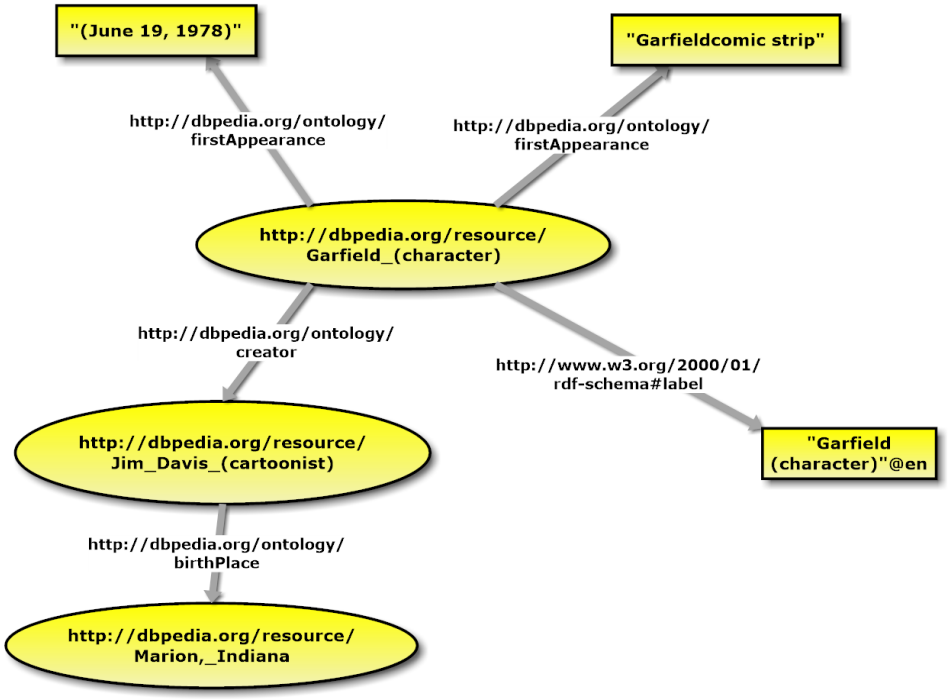
webnek. Szabályai megegyezik az XML leírás szabályaival. Azért ez terjedt el, mert sok XML olvasó alkalmazás létezik, amiket fel lehetett használni az adatok kezelésére, ám a felhasználók szempontjából viszont nem értelmezhető egyszerűen. Emiatt jelent meg a Turtle [107] formátum, amely a 3. példán látható. Itt az állításokat ugyanúgy hármasokként írhatjuk le, mint az N-triples-nél, de itt lehetőségünk van arra, hogy prefixeket alkalmazzunk. A prefixek arra szolgálnak, hogy az egyes URI névtereket ne kelljen kiírnunk. A példán láthatjuk, hogy a *dbo* a *<http://dbpedia.org/ontology>* névteret jelenti, ezáltal minden olyan helyen ahol *dbo*-t írunk, a rendszer ezt az URI-t fogja használni. Ezzel rövidebb és olvashatóbb azonosítókat tudunk készíteni. További lehetőség, hogy ha több állításunk van ugyanarról az alanyról, akkor nem kötelező minden állításban kiírni, hanem használhatjuk a ';'-t arra, hogy elválasszuk a állítmány-tárgy párosokat. A negyedik formátuma a szemantikus webnek az N3 [23], amely további lehetőségeket biztosít az adatok leírásában, de ezek ritkán vannak használva, így azokkal nem foglalkozunk.

Példa 1 Garfield példa N-triples formátumban

```
1: <http://dbpedia.org/resource/Garfield_(character)>
   <http://dbpedia.org/ontology/creator>
   <http://dbpedia.org/resource/Jim_Davis_(cartoonist)> .
2: <http://dbpedia.org/resource/Garfield_(character)>
   <http://dbpedia.org/ontology/firstAppearance> "(June 19, 1978)" .
3: <http://dbpedia.org/resource/Garfield_(character)>
   <http://dbpedia.org/ontology/firstAppearance> "Garfieldcomic strip" .
4: <http://dbpedia.org/resource/Garfield_(character)>
   <http://www.w3.org/2000/01/rdf-schema#label> "Garfield (character)"@en .
5: <http://dbpedia.org/resource/Jim_Davis_(cartoonist)>
   <http://dbpedia.org/ontology/birthPlace>
   <http://dbpedia.org/resource/Marion,_Indiana> .
```

1. Definíció. *(RDF Hármas) Legyenek I, B, és L (IRI-k, üres csúcsok, literálok) halmazok. Ekkor a $(v_1, v_2, v_3) \in (I \cup B) \times I \times (I \cup B \cup L)$ egy* RDF hármas*, ahol v_1 az alany, v_2 az állítmány és v_3 a tárgy. Az RDF hármasok véges halmazát nevezzük RDF gráfnak vagy RDF adathalmaznak.*

Az állításokat csoportosítani szoktuk aszerint, hogy a tárgy helyén egy másik objektum, vagy szöveges adat található. Azokat az állítmányokat, amelyek egy objektumra mutatnak, *Object property*-nek nevezzük, még azokat, amelyek szöveges adatra, *Data property*-nek. A szöveges adatoknál láthatunk egy '@' jelet, ami az adott szöveg nyelvét jelenti, jelen esetben ez az angol. Ezeknél az adatoknál lehetőségünk van ugyanazt a Data property-t több nyelven is megadni. A másik tulajdonsága ezeknek az állításoknak, hogy



2.2. ábra. Garfield példa gráf: szemantikus adatok tárolásának gráfos szemléltetése a Garfield karakter alapján. A szöveges csomópontok a Data property-k, az IRI-k az Object property-k. Egy Object Property-ből új élek indulhatnak, lásd: *Jim_Davis_(cartoonist)*

ha veszünk egy Object property-t, akkor a tárgy helyén szereplő objektumból szintén indulhat ki újabb állítás. Ezáltal az állítások egy gráfot képeznek. A példa gráf reprezentációját láthatjuk a 2.2. ábrán, ahol a *dbr:creator* egy olyan Object property, aminek a tárgya *dbr:Jim_Davis_(cartoonist)*, akiről egy újabb állítás létezik, hogy hol született.

2.1.2. SPARQL lekérdező nyelv

A szemantikus web használatának következő lépése az adatok lekérdezése. A W3C erre a SPARQL (SPARQL Protocol and RDF Query Language) [44] nyelvet készítette el. A lekérdezések a részgráf kereséshez hasonlóak. Egy tipikus SPARQL lekérdezés hármasmintákat tartalmaz. A változók behelyettesítése azok az objektumok lesznek, amelyek megfelelnek a hármasmintáknak. A 4. példa egy lekérdezést mutat be a 'Garfield' adatra, ahol szintén használhatjuk a prefixeket, hogy rövidebben tudjuk leírni az azonosítókat. A lekérdezésben a '?' karakterrel kezdődő szavak a változók, amelyek tárolni fogják az eredményt. A lekérdezés felépítése hasonló az SQL-hez: van egy SELECT rész, ahol megadjuk milyen változóknak az értékeit szeretnénk visszakapni, majd van egy WHERE, ahol megadjuk a feltételeket. Az itt leírt hármasokat nevezzük *hármasminták*nak, amik olyan

Példa 2 Garfield példa RDF/XML formátumban

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dbr="http://dbpedia.org/ontology/"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdf:Description rdf:about="http://dbpedia.org/resource/Garfield_(character)">
    <dbr:creator>
    <rdf:Description rdf:about="http://dbpedia.org/resource/Jim_Davis_(cartoonist)">
        <dbr:birthPlace rdf:resource="http://dbpedia.org/resource/Marion,_Indiana"/>
    </rdf:Description>
    </dbr:creator>
    <dbr:firstAppearance>(June 19, 1978)</dbr:firstAppearance>
    <dbr:firstAppearance>Garfieldcomic strip</dbr:firstAppearance>
    <rdfs:label xml:lang="en">Garfield (character)</rdfs:label>
    </rdf:Description>
</rdf:RDF>
```

Példa 3 Garfield példa Turtle formátumban

```
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbr: <http://dbpedia.org/resource/> .
dbr:Garfield_(character)
    dbo:creator dbr:Jim_Davis_(cartoonist) ;
    dbo:firstAppearance "(June 19, 1978)", "Garfieldcomic strip" ;
    rdfs:label "Garfield (character)"@en .
dbr:Jim_Davis_(cartoonist) dbo:birthPlace dbr:Marion,_Indiana .
```

hármasok, amelyekben szerepelhet változó is. Láthatjuk, hogy most három hármasmintát adtunk meg. Az első azt mondja meg, hogy a 'Garfield' azonosítóhoz van olyan állítás, ami 'creator' és ennek az állításnak a tárgyát tároljuk a *?creator* változóban. Ezenkívül ugyanerre az alanyra azt is mondjuk, hogy létezik 'label' állítás is, aminek a tárgyát a *?title* változóban tároljuk. A készítőre egy újabb állítást fogalmazunk meg, hogy van neki születési hely információja, amit letárolunk a *?place* változóban. Végül egy szűrési feltételt adunk meg, hogy csak azok a *?title* értékek érdekelnek minket, amelynek a nyelve angol. A LIMIT pedig leszűkíti az eredményeink számát 100-ra. Ha valamely állítás nem létezik az adathalmazban, akkor nem kapunk eredményt. Ám vannak olyan esetek amikor szeretnénk visszakapni részeredményeket is. Ha van egy olyan adattagunk, amely nem mindig létezik, de szeretnénk a többi változó eredményét akkor is megkapni, akkor használhatjuk az OPTIONAL kulcsszót, melyben megadott feltételeknek a teljesülése nem kötelező.

A SPARQL-ben nem csak SELECT lekérdezés írására van lehetőségünk, hanem négy

Példa 4 Garfield SPARQL lekérdezés példa

```
1: PREFIX dbo: <http://dbpedia.org/ontology/>
2: PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3:
4: SELECT ?title ?place WHERE {
5:       <http://dbpedia.org/resource/Garfield_(character)> dbo:creator ?creator ;
6:           rdfs:label ?title .
7:       ?creator dbo:birthPlace ?place .
8:       FILTER(LANG(?title) = "en")
9: } LIMIT 100
```

lekérdezés típus található, amelyből a SELECT az első. Ezenkívül futtathatunk ASK lekérdezéseket, amelyek egy igaz/hamis értékkel térnek vissza az alapján, hogy az adott adathalmazban van-e olyan részgráf, amit a WHERE feltételben megadtunk. Használhatunk CONSTRUCT lekérdezéseket, amelyek új adathalmazt hoznak létre. A lekérdezés WHERE feltételében megadjuk azokat a hármasmintákat, amiket keresünk a gráfban. A CONSTRUCT résznél pedig megadjuk, hogy azokból a változókból, hogyan készüljenek állítások. A negyedik pedig a DESCRIBE, amely egy adott objektumról ad vissza információt. A SPARQL 1.1 óta megjelentek az adatmódosító lekérdezések (INSERT, DELETE, UPDATE) is, de ezeket a legtöbb lekérdező felület nem támogatja.

Az alábbi definíciók a SPARQL formális leírása, amelyet Pérez írt le cikkében [42].

2. Definíció. *Legyen V különböző változók halmaza az $(I \cup B \cup L)$ felett. Ahol az I az IRI-k halmaza, a B az üres csúcsok halmaza, és az L a literálok halmaza. A változókat jelöljük kérdőjelekkel. Legyenek $?X, ?Y \in V$ változók és $c, d \in (L \cup I)$ literálok és IRI-k. A szűrőfeltételeket rekurzívan fogalmazzuk meg. Az $?X = c$, $?X = ?Y$, $c = d$, $bound(?X)$, $isIRI(?X)$, $isLiteral(?X)$, és $isBlank(?X)$ atomi szűrőfeltételek. Ezután, ha R_1, R_2 szűrőfeltételek, akkor $\neg R_1$, $R_1 \wedge R_2$ és $R_1 \vee R_2$ is szűrőfeltételek.*

3. Definíció. *Egy SPARQL kifejezés rekurzívan épül fel a következőképpen:*

1. *$t \in (I \cup V) \times (I \cup V) \times (L \cup I \cup V)$ egy SPARQL kifejezés,*
2. *ha Q_1, Q_2 SPARQL kifejezések, és R egy szűrőfeltétel, akkor Q_1 Filter R, Q_1 Union Q_2, Q_1 Opt Q_2, és Q_1 And Q_2 is SPARQL kifejezések.*

Az alábbi definíciókat Schätzle definíciói [122] alapján egészítik ki a formális leírást.

4. Definíció. *(Hármasminta, Alap gráf minta) Legyen V (változók) egy véges halmaza az I és L halmaznak, ekkor egy $t \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$ hármas egy hármasminta. A hármasminták véges sorozata az alap gráf minta.*

5. Definíció. *(Változó behelyettesítés) Legyen q egy alap gráf minta. Egy változó behelyettesítés egy leképezés a q változóiból, $var(q) = \{v|\exists(s,p,v) \in q \text{ or } \exists(v,p,o) \in q\}$, az IRI-k literálok halmazára, ami $r : var(q) \to (I \times L)$.*

6. Definíció. *(Lekérdezés eredmény) Legyen q egy SPARQL lekérdezés, ami egy alap gráf minta és legyen T egy RDF adathalmaz. A lekérdezés eredménye a változók behelyettesítésének halmaza. $R = \{r : var(q) \to (I \times L)\}$, feltéve, hogy $apply(r,q) \subseteq T$, ahol $apply(r,q) = \{(s,p,o)|(\exists(v,p,o) \in (V \times I \times (I \cup L)) : (v,p,o) \in q \wedge (r(v),p,o) = (s,p,o)) \vee (\exists(s,p,v) \in (I \times I \times V : (s,p,v) \in q \wedge (s,p,r(v)) = (s,p,o))\}$.*

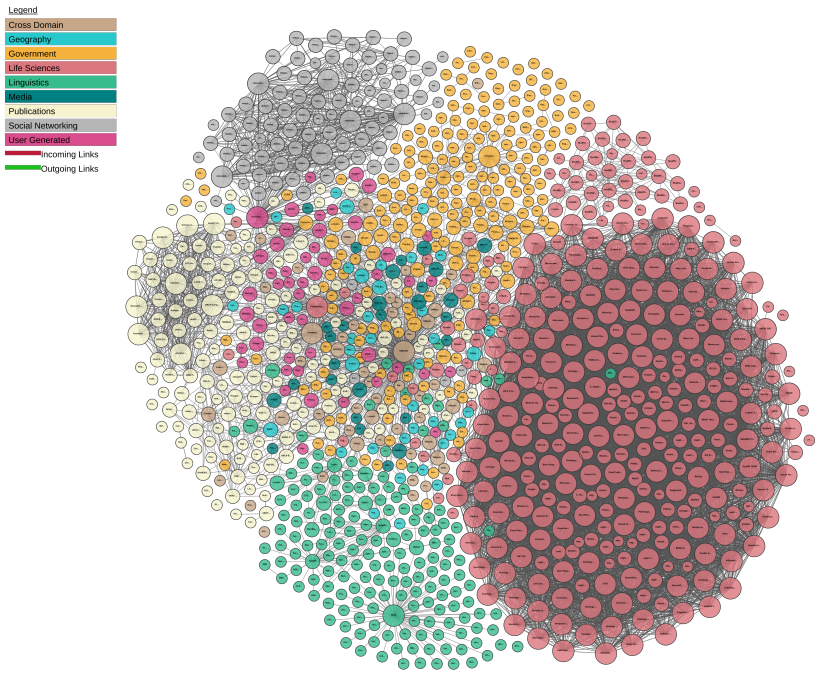
2.2. LOD felhő, SPARQL végpontok

A szemantikus web elgondolás alapja, hogy az adatok elérhetőek legyenek a világhálón és összekapcsolhatóak legyenek más adathalmazokkal is. A LOD (Linked Open Data) [66] célja, hogy ezeket az adathalmazokat összefogja. A 2.3. ábrán láthatjuk a LOD felhőt, amely az elérhető adathalmazokat és a köztük lévő kapcsolatokat reprezentálja. Ahhoz, hogy egy adathalmaz bekerüljön ebbe a diagramba a következőknek kell megfelelnie: elérhetőnek kell lennie egy URL-en keresztül, RDF formátumot kell használnia, legalább 1000 hármassal kell rendelkeznie, legalább 50 kapcsolattal kell kapcsolódnia más adathalmazhoz, és végül az elérést biztosítani kell RDF crawler, RDF dump vagy SPARQL végpont segítségével. A SPARQL végpontok olyan alkalmazások, amelyek lehetőséget biztosítanak egy API-n keresztül, hogy SPARQL lekérdezéseket futtassunk rajtuk. A diagramon szereplő adathalmazok méretét jellemzi a körök nagysága, a színek pedig az egyes tématerületeket jelentik. A képnek elérhető egy interaktív verziója is, ahol meg tudjuk tekinteni az egyes köröket részletesebben. A dolgozatban említettem az adathalmazok méretét. A LOD felhőről leolvashatjuk például, hogy a DBPedia 9,5 milliárd, a Freebase 337 millió vagy a LinkedMDB 6 millió hármassal rendelkezik. Ezek a számok már magukban is elég méretűek ahhoz, hogy kihívást jelentsen a hatékony szemantikus lekérdezésük.

Amennyiben olyan információra van szükségünk, amely több végponton is szerepel, akkor a SPARQL 1.1 [111] óta használhatjuk a SERVICE kulcsszót. Ezeket a lekérdezéseket hívjuk federált lekérdezéseknek, melyet szemléltet az 5. példa, amely a FedBench benchmark lekérdezés halmaz Cross-Domain[5] lekérdezéséből származik. A lekérdezés a DBpediáról [46] kérdezi le az olasz rendezők filmjeit, majd a LinkedMDB [68] végpontról

[4]Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/

[5]http://fedbench.fluidops.net/resource/fbenchQuery:cross-domain_5



2.3. ábra. LOD felhő: Linked Open Data adathalmazok és kapcsolatuk gráfja. Ezt az ábrát nevezik LOD felhőnek[4]. A szinek az adathalmaz tematikáját, a körök méretei az adathalmaz méretét reprezentálják.

lekérdezi a filmek műfajait. Láthatjuk, hogy a SERVICE kulcsszónál meg kell adnunk a SPARQL végpont URL-jét is.

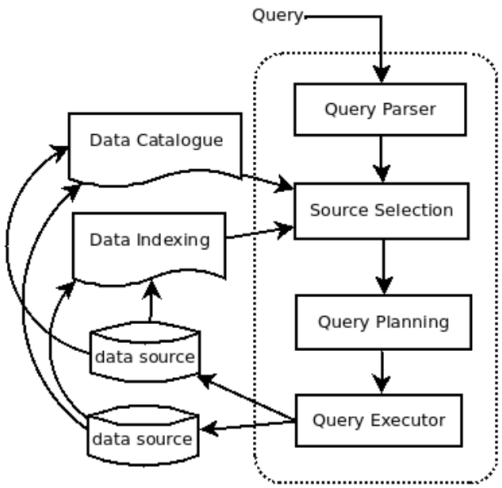
2.3. Federált rendszerek

A federált lekérdezések fő problémája, hogy a legtöbb végpont nem támogatja őket. Aminek az az oka, hogy egy ilyen lekérdezés lefuttatása költséges. Ha vesszük az előbbi példát, akkor a DBpediának kell az al-lekérdezést elvégeznie a LinkedMDB végponton, a részeredményeket le kell tárolnia, majd összekapcsolnia azokat.

A felhasználók szempontjából se egyszerű használni ezeket, hisz tudniuk kell, hogy mely adat mely végponton érhető el és ismerniük kell a végpont URL-jét is. Emiatt keletkeztek olyan rendszerek, amelyek ezen tudnak segíteni. Ezeket nevezzük federált rendszereknek. Egy jó összefoglalást készített Rakhmawati a szerzőtársaival [108] ezekről a rendszerekről. A 2.4. ábrán az általános felépítését és működését láthatjuk egy federált rendszernek. Az ilyen rendszerek fő komponensei a lekérdezés értelmező, a végpont választó, a lekérdezés ütemező és maga a végrehajtó folyamat. A lekérdezés értelmező kinyeri a hármasmintákat és a különböző feltételeket a lekérdezésből. Ezután a végpont

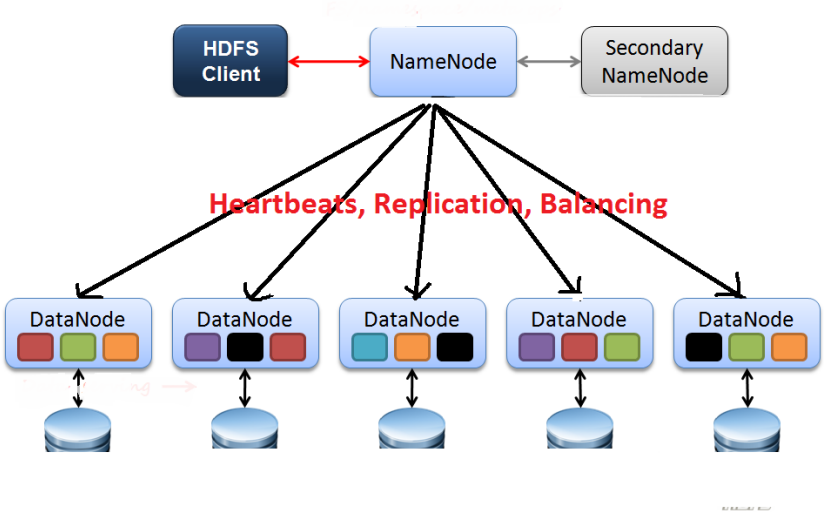
Példa 5 Federált lekérdezés: Olasz rendezők filmjei

```
1: PREFIX owl: <http://www.w3.org/2002/07/owl#>
2: PREFIX linkedmdb: <http://data.linkedmdb.org/resource/movie/>
3: PREFIX dbo: <http://dbpedia.org/ontology/>
4: PREFIX DBR: <http://dbpedia.org/resource/>
5:
6: SELECT ?film ?director WHERE {
7:       ?film dbo:director ?director .
8:       ?director dbo:nationality dbr:Italy .
9:       SERVICE <http://www.linkedmdb.org/sparql> {
10:          ?x owl:sameAs ?film .
11:          ?x linkedmdb:genre ?genre
12:      }
13: } LIMIT 10
```



2.4. ábra. Federált rendszer felépítése [108]: A federált lekérdezés feldolgozásának lépései: 1- Query Parser: értelmezi és részekre bontja a lekérdezést. 2- Source Selection: a hármasmintákhoz kiválasztja a megfelelő végpontokat. Ez történhet egy katalógus alapján, vagy ASK lekérdezések segítségével. 3- Query Planning: lekérdezési terv készítését végzi a gyűjtött információk alapján. 4- Query Executor: Lefuttatja az al-lekérdezéseket és a részeredményeket összekapcsolva, elkészíti a végső eredményt.

kiválasztó eldönti, hogy az adott hármasmintát mely végpont tudja megválaszolni, melynek eldöntésére vagy valamilyen katalógust használ, vagy ASK lekérdezések segítségével dönti el, hogy az adott végpont képes-e válaszolni. Ha minden hármasmintához megvannak a végpontok, akkor a lekérdezés ütemező al-lekérdezéseket készít belőlük, az alapján, hogy mely végpontra kell őket küldeni. Ezután a végrehajtó lefuttatja a lekérdezéseket és összekapcsolja a kapott eredményeket. A dolgozatban a végpont választási stratégiával fogok foglalkozni.

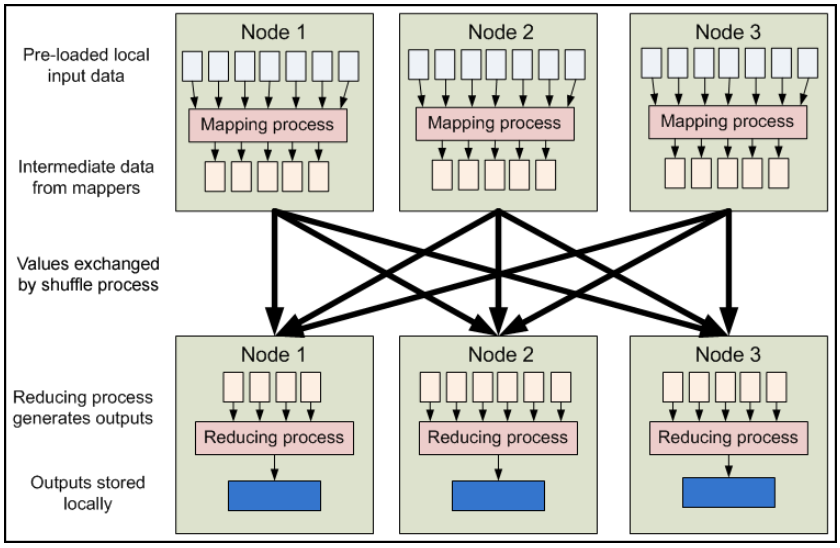


2.5. ábra. HDFS architektúra[7]: A hadoop elosztott fájlrendszerének komponensei: 1- DataNode: tárolja az adatokat, hozzáférést biztosít a *chunk*-okhoz. 2- NameNode: metadatokat tárol a HDFS-en tárolt fájlok darabjairól. 3- Secondary NameNode: másodlagos NameNode, ami átveszi az elsődleges szerepét, ha az meghibásodik.

2.4. Osztott adattárolás és számítás

A technológiai fejlődés miatt egyre több adatot generálunk nap mint nap. Ezeknek az adatoknak a kezelésére új rendszerekre, architektúrákra van szükségünk. Az új architektúrák már nem egy gépen futnak, hanem klaszterbe kötött gépeken, ezáltal jobb a skálázhatósága a rendszernek. Ha több erőforrásra van szükségünk, akkor csak újabb gépeket kell bekapcsolnunk a klaszterbe. Ezek az új rendszerek azt se követelik meg, hogy az összekapcsolt eszközök egy architektúrájúak legyenek. Lehetnek olyan asztali gépek összekapcsolva, amelyek különböző memóriával rendelkeznek. A legismertebb ilyen eszköz a Hadoop, amely egy open-source eszköz. Ez a rendszer rendelkezik egy elosztott fájlrendszer technikával a HDFS-sel (Hadoop Distributed File System) [77], amelyet a legtöbb Big Data eszköz használ. Ez két komponensből tevődik össze. Az első a NameNode, amely metainformációkat tárol a klaszterben található fájlokról, a másik a DataNode, amely tárolja az adatot. Az adatok tárolása úgy működik, hogy amikor felmásolunk egy állományt a fájlrendszerre, akkor azt a rendszer úgynevezett chunk-okra darabolja, amelyek általában 64 MB vagy 128 MB méretűek, és szétosztva tárolódnak a klaszterben. A HDFS előnye más elosztott adattárolásokkal szemben, hogy hiba-toleráns. Ezt úgy oldja meg, hogy minden a klaszterben tárolt chunkból több példány (replika) található. Ez tipikusan három példányt jelent, ami három különböző gépen tárolódik. Ha egy gép meghibásodik,

[7]HDFS architektúra: http://dailyhadoopsoup.blogspot.hu/2014/01/hdfsmoving-computation-is-cheaper-than.html



2.6. ábra. MapReduce folyamat[9]: 1- Párhuzamosan a gépeken elindulnak a Map folyamatok a *chunk*-okon, amelyek kulcs-érték párokat készítenek. 2- A kulcs-érték párok csoportosítása kulcs alapján, majd a csoportokat elküldeni más gépekhez. 3- Az egyes csoportokon párhuzamosan lefutnak a Reduce folyamatok.

akkor sincs adatvesztés, mert a másolatok még használhatóak. A 2.5. ábrán láthatjuk a chunkokat különböző színekkel.

Erre fájlrendszerre épül az elosztott számítási modell a MapReduce, amelyet a Google fejlesztett ki [58]. A működése hasonló az "oszd-meg-és-uralkodj"-elvhez. Ennek a modellnek két fázisa van: a Map és a Reduce. Egy alkalmazás elkészítésénél azt kell megadnunk, hogy mit csináljanak az egyes folyamatok az adattal. Nem kell azzal foglalkoznunk, hol van az adat, vagy hogy hogy juttatjuk el egyik folyamattól a másikhoz. A 2.6. ábrán láthatjuk, hogyan működik a rendszer. A Map folyamatok párhuzamosan futnak le az egyes gépeken és dolgozzák fel az adatot soronként. A cél, hogy ott fussanak a folyamatok, ahol az adat van, és ne kelljen mozgatni az adatot a klaszterben. A Map folyamatok kimenete egy kulcs-érték pár lesz, amelyet mi határozunk meg. Ezután jön a *shuffle* fázis, amely a kulcs alapján szétosztja az értékeket a Reducer folyamatok között. Egy Reducer mindig egy kulcsot és a hozzá tartozó összes értéket kapja meg. A feldolgozás után az eredményeket a Reducerek visszaírják a HDFS-re. Egy Map és Reduce folyamat lefutását nevezzük *Job*nak.

Ennek a modellnek a hátránya, hogy sokat használja a háttértárat a feldolgozás alatt. Egy Job lefutása közben akár 4-5-ször is kiírja és olvassa a részeredményeket, emiatt készült egy új rendszer, amely már nem a merevlemezt használja adattárolásra, hanem a klaszterben lévő gépek memóriáját. Ez a rendszer a Spark [78], amely RDD [99] ob-

[9]MapReduce folyamat: https://wikis.nyu.edu/download/attachments/74681720/DataFlow.png

jektumokat használ az adatok tárolására. Ezek az objektumok elosztva tárolódnak és dolgozódnak fel a klaszter memóriájában, és a felhasználó szempontjából egy objektumként kezelhetőek. Mivel csak a futás elején használja a háttértárat, emiatt gyorsabban tud működni mint a Hadoop MapReduce. Másik előnye a rendszernek, hogy rendelkezik optimalizációval, amely az alkalmazás lefutása előtt történik meg. A MapReduce algoritmusnál az optimalizáció a mi feladatunk, vagyis a Jobok megfelelő sorrendjének meghatározása. A Spark viszont ellenőrzi az adatok elérhetőségét, és hozzáférhetőségét. Az összegyűjtött információk alapján egy kéréssel ekvivalens sorrendet tud előállítani a Joboknak, ami szintén gyorsítja a lefutást.

2.5. Osztott gráf tárolás és feldolgozás

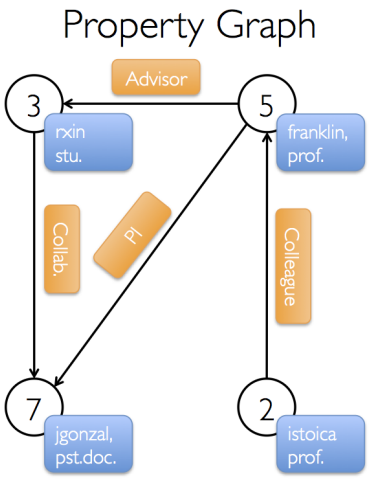
2.5.1. Tulajdonság gráfok (Property graph)

A Spark rendszer további előnye, hogy rendelkezik különböző kiegészítő könyvtárakkal, amelyek speciális adatok elemzésére szolgálnak. Ilyen adatok lehetnek a stream jellegűek, vagy épp a gráf jellegűek. A gráfok tárolása a klaszterben nem egyértelmű dolog, hisz a gráf egy összefüggő adatszerkezet. Ha a csúcsokat egyenletesen tároljuk a klaszterben akkor a köztük lévő élek miatt sok lesz a kommunikáció a gépek között. Ha a kisebb csoportokat tároljuk az egyes gépeken, akkor a szomszédok elérése gyorsabb lehet, viszont a gépek terhelése nem lesz egyenletes. A legtöbb gráftároló alkalmazás a minimális vágás módszerét alkalmazza. Ezzel szemben a Spark GraphX a csúcsok mentén választja szét a gráfot, vagyis van olyan csúcs, amely több gépen is megtalálható. Az adatokat úgynevezett tulajdonság gráfban (Property graph) tárolja, amely olyan gráf, amelyben mind a csúcsokhoz, mind pedig az élekhez információt rendelhetünk. A 2.7. ábrán láthatunk egy példát, ahol a csúcsok a személyek, az élek pedig a köztük lévő kapcsolatok. A csúcsoknak név és foglalkozás információi vannak, az éleknek pedig a kapcsolatuk típusa.

Az alábbi definíciók a Schätzle és társai által bemutatott formális modell [122] a tulajdonság gráfokra.

7. Definíció. *(Tulajdonság gráf) Legyen $PG(P) = (V, E, P)$ egy tulajdonság gráf, ahol V a csúcsok halmaza, E az élek halmaza és P tulajdonságok a gráfban. $P_V(i)$ jelentse az i csúcs tulajdonságait és $P_E(i, j)$ az (i, j) él tulajdonságait. Egy adott tulajdonság jelölése $i.x$ és $(i, j).x$.*

Egy RDF halmazból készített tulajdonság gráfot a következőképpen írjuk le. Legyen



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

2.7. ábra. Tulajdonság gráf (property graph) [11]: A gráfok éleihez és csomópontjához tetszőleges számú tulajdonságot tudunk rendelni.

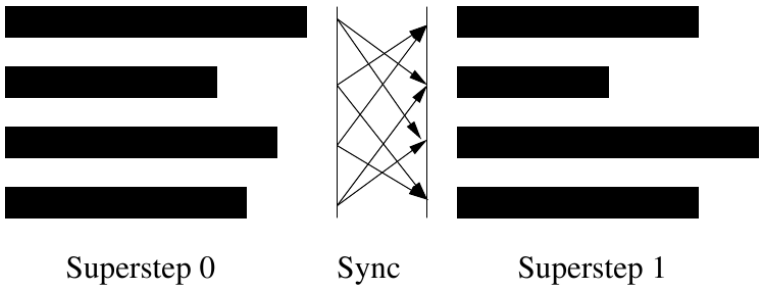
$V = \{s|\exists(s,p,o) \in T\} \cup \{o|\exists(s,p,o) \in T : \wedge\, o \in I\}$ és $E = \{(s,o)|\exists(s,p,o) \in T : o \in I\}$ a csúcsok és élek halmaza. Megjegyezzük, hogy a párhuzamos élek megengedettek. Minden $i \in V$ csúcsra, legyen $P_V(i) = \{o|\exists(s,p,o) \in T : o \in L \wedge s = i\}$. Minden élre $(i,j) \in E$, legyen $P_E(i,j) = p$, ha az él az $i,p,j \in T$ hármasból készült. Ahogy látható különbséget teszek a Data és Object property-k között. A Data property-k a tulajdonságai lesznek a csúcsoknak, még az Object property-k élek lesznek a gráfban.

2.5.2. Bulk Synchronouse Parallel (BSP) modell

Az osztott gráf feldolgozás alapja a Bulk Synchronouse Parallel (BSP) [22] modell, amelyet a processzor magok párhuzamosításához fejlesztettek ki. A feldolgozás lényege, hogy a processzorok párhuzamosan számolnak, majd az eredményeiket elküldik a többi processzornak. Ekkor van egy szinkronizációs fázis, hogy minden processzor befejezze az adott számítását és elküldje a részeredményét. Amikor ez megtörtént, akkor indul a következő számítási fázis. Ezt a folyamatot látjuk a 2.8. ábrán.

Az elosztott gráf feldolgozás algoritmusa hasonlóan működik. Egy tipikus gráffeldolgozás iterációkból áll és a szomszédos csúcsok információ alapján számítódnak az értékek. Tehát a processzorok helyét átveszik a csúcsok, az üzenetek küldése pedig az éleken keresztül történik. Erre a programozási modellre szokták mondani, hogy gondolkozz úgy mint egy csúcs ('think like a vertex'). Egy ilyen algoritmus elkészítésénél három folyamatot kell megvalósítani. Az első folyamat magának a csomópontoknak az algoritmusa. Egy csomópont rendelkezik egy saját értékkel és kap egy új értéket, ami alapján kiszámolja az

[11] Property graph: http://spark.apache.org/docs/latest/img/property_graph.png



2.8. ábra. Bulk Synchronouse Parallel (BSP) modell [13]: Párhuzamosított feldolgozás folyamata, ahol a feldolgozó egységeknek kommunikálniuk kell a feldolgozás során. 1- Egy adott feldolgozó egység számítást végez a lokális és kapott adatain. 2- Az eredményeit elküldi a megfelelő folyamatoknak. 3- Amikor minden egység végzett a számításával és elküldte az eredményét, előröl kezdődik a folyamat.

új, saját értékét. A második folyamat az üzenetek küldése. Meg kell adnunk, hogy mikor, melyik élen, milyen üzenet kerüljön elküldésre. A harmadik folyamat pedig az üzenetek összekapcsolása. Az első folyamatnak egy új üzenetet kell kapnia, ami az összes szomszéd által küldött üzenetnek a redukáltja. Ezt a redukciót nekünk kell megvalósítani. Egy ilyen folyamat addig fut, amíg van üzenet a gráfban, vagy beállíthatunk egy fix iterációszámot.

A Spark GraphX úgy működik, hogy mindig az aktív éleken megy végig és az éleknél kell eldönteni, hogy küldünk-e üzenetet az adott élen. Egy él akkor aktív, ha valamelyik csomópontja kapott üzenetet az előző iterációban. Ha nincs üzenet a gráfban, akkor egyik él se lesz aktív és a folyamat leáll. A folyamat első lépése egy inicializációs üzenet, amelyet minden csomópont megkap, emiatt az első iterációban minden él aktív lesz.

2.6. Biszimuláció

A biszimuláció a csúcsok összevonását jelenti. Ahhoz, hogy ezt meg tudjuk tenni, be kell vezetnünk pár definíciót. A 8. definíció megfogalmazza a hasonlósági relációt a csúcsok egy halmazára. A definíció Herzinger korábbi cikkén [21] alapul, ahol a szerzők egy hasonlósági relációt mutatnak be címkézett gráfokon.

8. Definíció. *(szimuláció) Egy bináris reláció '$\leq$' $\subseteq V^2$ csúcsok halmaza felett egy szimuláció, ha $u \leq v$ minden $e = (u, w) \in E$ és $e' = (v, w) \in E$ élre úgy, hogy $label(e) = label(e')$. A v csúcs szimulálja az u csúcsot ha létezik szimuláció ($\leq$) és $u \leq v$ között. Az u és v csúcsok hasonlóak, jelöljük $u \sim v$, ha u szimulálja v-t és v szimulálja u-t.*

[13] BSP modell: http://www.multicorebsp.com/images/algorithm.gif

A későbbi fejezetben olyan algoritmust mutatunk be, ahol *lánc*-ként és *nem-lánc*-ként jelölünk meg éleket, és ezek alapján készítjük el a szemantikus gráf struktúráját. Az algoritmusban szükség van a csúcsok összevonására, amelynek a feltételét a 9. definíció mutatja be. Két csúcs összevonható, ha hasonlóak és ugyanazon kimenő élekkel rendelkeznek, amelyek meg vannak jelölve *nem-lánc* tulajdonsággal.

9. Definíció. *(összevonható csúcsok). Legyen $u, w \in E$ két csúcsa a $G = (V, E)$ szemantikus gráfnak. u és v összevonható, ha*

1. *$u \sim v$ és*

2. *u és v minden kimenő éle* nem-lánc*-ként van megjelölve.*

Amikor két csomópontot összevonunk, akkor az új csúcs címkéjét lecseréljük egy egész értékre, ami az összevont csúcsok számát jelenti. Ehhez a következő leírást alkalmazzuk, ahol $u, v \in V$ csúcsokat vonjuk össze a w csúcsba:

$$value(w) := \begin{cases} 2, & \text{ha } u \text{ és } v \text{ nem összevont csúcs} \\ \text{value(u)+1}, & \text{ha } u \text{ összevont csúcs, de } v \text{ nem} \\ \text{value(v)+1}, & \text{ha } v \text{ összevont csúcs, de } u \text{ nem} \\ \text{value(u)+} & \\ \text{value(v)}, & \text{ha } u \text{ és } v \text{ is összevont csúcs} \end{cases} \tag{2.1}$$

3. fejezet

Szemantikus adatok kliens-szerver architektúrán

3.1. Bevezető

A szemantikus web elgondolásának fő problémája a technikai megvalósítás. Láthattuk, milyen formában tároljuk az adatokat szemantikusan, láthattuk, hogyan tudjuk azokat lekérdezni. Arról viszont nem esett szó, hogy hogyan tároljuk és dolgozzuk fel őket hatékonyan. Mivel az adatok strukturálatlanok, emiatt a lekérdezések elvégzése számításigényes feladat. Ha olyan eszközöket veszünk, amelyeknek a számítási kapacitása kicsi, akkor azok az eszközök nem tudják elérni ezt a nagy tudásbázist. A legegyszerűbb példa ilyen eszközökre a mobil készülékek, melyek napjainkban sokat fejlődtek, és egy-két készülékben néha több processzor mag van, mint egy asztali számítógépben, ami elég ahhoz, hogy bonyolult számításokat végezzünk. A probléma a rendelkezésre álló memória és tárhely. A szemantikus adatok tipikusan nagy méretű, akár több millió hármasból álló adathalmazok. Ahhoz, hogy ezt kezelni tudjuk más architektúrában kell gondolkodnunk a mobilok számára. Ezzel a témával foglalkoztam a diplomamunkámban [1] is. Erre egy jó megoldás a kliens-szerver architektúrák [24], amelyek arra lettek kitalálva, hogy olyan kliensek is képesek legyenek használni bizonyos alkalmazásokat, amelyekhez nincs elegendő erőforrásuk. A régi időkben a kliens egy asztali számítógép volt, amely gyengébb processzorral, kevesebb memóriával rendelkezett, mint amire szüksége lett volna. A szerver viszont olyan erőforrásokkal rendelkezett, amellyel könnyen el lehetett végezni, akár több számítást is. Az architektúra úgy működött, hogy a kliens elküldte a számítás igényes feladatot a szervernek, az kiszámolta és visszaküldte a kliensnek. Az ilyen számítási feladatok lehettek akár adatbázisrendszerek lekérdezései is. Egy kisebb adatbázist a kliens könnyen tud ke-

zelni, de ha már túl nagy, akkor már hatékonyabb egy szerveren tárolni és a szerverrel számoltatni ki a lekérdezéseket. A szemantikus web egy nagy adatbázis (tudásbázis), melyet a legtöbb kliens nem tud kezelni. Ahhoz, hogy le tudjanak kérdezni, szintén alkalmazhatjuk a kliens-szerver architektúrát.

Ebben a fejezetben bemutatok három olyan rendszert, amely ezen az architektúra elgondoláson alapszik. Az első két megoldás olyan rendszerek, ahol saját tudásbázisunkat fogjuk alkalmazni, még a harmadik rendszer kibővíti ezt a világhálón felelhető tudásbázisokkal. Az itt bemutatott alkalmazások csoportos eredmények, melyeket Rácz Gáborral, Pinczel Balázzsal és Matuszka Tamással készítettünk. Az én fő hozzájárulásom a szerver oldalon történő lekérdezések készítése valamint a kommunikációs protokoll a kliens és a szerver között.

Vegyük sorra az itt bemutatni kívánt rendszereket. Az első rendszer egy ipari kutatás-fejlesztési projekt keretein belül készült el. A motiváció az volt, hogy a vállalatoknak a mindennapokban rengeteg dokumentumot kell kezelniük. Ezek lehetnek egyszerű szövegek, önéletrajzok, projekt beszámolók, weboldalak, e-mailek, blog bejegyzések, stb., de az ezekben tárolt információk és a köztük lévő kapcsolatok ugyanúgy fontosak lehetnek. Ebből jött az elgondolás, hogy ha a probléma ugyanaz egy vállalatnál, mint amiért a szemantikus web elkészült, akkor miért ne használjuk az ott alkalmazott technikákat itt is. Ha az adatok olyan formában lennének tárolva, amelyek összekapcsolhatóak, akkor a következtetés segítségével akár új információkat is képesek lehetünk kinyerni. Másfelől, ha az adatok olyan formában tárolódnak, amelyek összekapcsolhatóak a Linked Open Data-val, akkor kiegészíthetőek a világhálón található információkkal is. A fejezetben egy ilyen prototípus alkalmazást mutatok be, ahol egy vállalat adatait a szemantikus web szabványaival tároljuk. Egy vállalat számára fontos, hogy az adatokat ne mindenki érje el, emiatt különböző hozzáférési szinteket alkalmaztunk. Másik szempont, hogy a SPARQL-ben nem jártas felhasználók is használni tudják, emiatt minden művelet űrlapokon keresztül történik. Végül pedig, hogy az adatokat bárhonnan el lehessen érni a kliens alkalmazás mobil eszközökre lett tervezve. Az adatok tárolását és a költséges számításokat egy szerver végezte. Ebben a cikkben a kliens szerver kommunikáció, a dinamikus adat továbbítás a kliens felé, a lekérdezések generálása és futtatása köthető hozzám.

A második alkalmazás egy beltéri navigációs rendszer, amely szemantikus technológiákat használ a navigációhoz. A beltéri navigáció kutatás egy aktív kutatási terület napjainkban, hisz sokszor kerülünk olyan helyzetbe, hogy meg akarunk találni valamit egy épületen belül, mint például áruházakban, bevásárlóközpontokban, raktárakban, egyetemeken, múzeumokban, stb. A kutatás fontosságát az is mutatja, hogy az ipar is foglalkozik a

témával. 2012 őszén több nagy cég fogott össze, hogy megalkossanak egy egységes beltéri navigációt[1]. Mi is a probléma a beltéri navigációval? A kültéri navigációt a Föld körül 20200 kilométerre keringő műholdak biztosítják, melyeknek a száma folyamatosan változik, hisz újabb és újabb technológiával rendelkezőket állítanak Föld körüli pályára. 2016 októberében 31 működő műhold biztosította a navigációt[2]. A műholdak keringési pályája úgy van kitalálva, hogy a Földön mindig legalább 4 műholdat látniuk kell a vevő egységeknek. A navigáció úgy működik, hogy ezek a műholdak rendszeres időközönként sugározzák a földre a pozíciójukat. A vevő egység a látott műholdak pozíciói alapján háromszögeléssel [17] megállapítja a saját pozícióját. Minél több műholdat lát az eszköz annál pontosabban tudja meghatározni a helyet. A probléma ezzel a rendszerrel az, hogy a jelek csak kültéren foghatóak. Ha egy épületen belül szeretnénk navigálni, akkor a műholdakat nem használhatjuk, emiatt kell más megoldásokat alkalmaznunk. Korábbi munkákban a pozíciót infravörös jelből [20, 25], különböző vezeték nélküli rendszerek jelerősségéből (GSM (Global System for Mobile Communications), Bluetooth, WiFi [37, 50, 84, 100]), a felhasználó mozgásának követéséből [53] vagy egy kamera képének feldolgozásából [51, 63] próbálták meghatározni. A kutatásunk célja az volt, hogy olyan beltéri navigációs rendszert készítsünk, amely a mobil eszközökben található szenzorokat, valamint a kiterjesztett valóságot alkalmazza a navigációhoz. A kiterjesztett valóság egy olyan eszköz, amely a kamera képét egészíti ki virtuális objektumokkal. A navigációnkban ezek az objektumok nyilak, melyek megmutatják a felhasználónak a helyes irányt. Egy épület leírásához a szemantikus web technológiát alkalmaztuk, mely lehetőséget biztosított nekünk arra, hogy minden objektumot megfelelően le tudjunk írni és SPARQL lekérdezésekkel el tudjuk készíteni az útvonalat. Az eredményül kapott adathalmazt a mobil készülékek már könnyen tudták alkalmazni a navigáció során. Ebben a kutatásban Matuszka Tamással dolgoztam együtt, ahol az én részem a szerver oldali feladatok (útvonal adatok készítése, adatok kezelése, kommunikáció a klienssel) voltak.

Az előző alkalmazásokban saját tudásbázist alkalmaztunk a kliens-szerver architektúrában, ezzel szemben a harmadik alkalmazásban az adatforrás a Linked Open Data, azaz a LOD felhő. Rengeteg szemantikus adatbázis található, amelyek különböző területek adatait tárolják. Ilyen területek például az informatika, a biológia, a kémia, a társadalomtudomány, stb.. A Linked Open szervezetnek a célja, hogy az adatok elérhetőek és összekapcsolhatóak legyenek más adathalmazokkal. Tehát minden adathalmaznak tartalmaznia kell olyan azonosítót, amellyel összekapcsolható más tudásbázissal [66]. Ezáltal egy nagyobb, egységes tudásbázis hozható létre. Ezek az azonosítók jellemzően URL-k,

[1] http://www.ravepubs.com/indoor-navigation-nokia-samsung-sony-join-forces/
[2] http://www.gps.gov/systems/gps/space/

amelyeket meg lehet nyitni egy egyszerű böngészőben. Ha megnyitunk egy ilyen oldalt, akkor egy úgynevezett szemantikus böngészőt kapunk, ami olyan formában jeleníti meg a szemantikus adatokat, hogy az könnyen értelmezhető legyen az emberek számára. Két kategóriája ismert. Az egyik gráfként jeleníti meg az RDF gráfot és a kapcsolatokat az egyes erőforrások között. A másik szöveges megjelenítést használ, ahol az állításokat táblázatos formában látjuk, a képek URL-je helyett képet látunk, és minden állítás kattintható, ezáltal át tudunk menni más erőforrások oldalára is. A szemantikus adatok terjedésével már több ilyen böngésző is készült [40, 47, 56], de jellemzően ezek a böngészők nem használhatóak mobil készülékeken. A célunk az volt, hogy egy olyan szemantikus böngészőt hozzunk létre, amely mobil készülékeken is használható. A rendszer lelke egy olyan kliens-szerver architektúra, ahol a szerver le tud kérdezni több SPARQL végpontot is. A legtöbb szemantikus böngésző csak a saját tudásbázisát képes megjeleníteni. Ha egy olyan erőforrásra kattintunk rá, amely nem az ő tudásbázisában szerepel, akkor átkerülünk egy másik szemantikus böngészőbe. Az elkészült alkalmazásunk viszont összegyűjti az összes információt és megjeleníti a mobil eszközön. A kutatásban szintén Matuszka Tamással dolgoztam együtt. A fő eredményem egy olyan szerver oldal készítése, amely képes több SPARQL végpontot elérni, és az ott található adatokat összegyűjteni. A kliensnek ezután csak meg kellett jelenítenie a kapott adatokat.

3.2. Kapcsolódó munkák

3.2.1. Szemantikus mobil alkalmazások és megjelenítők

A szemantikus technológiák használata széleskörű. A *'The semantic web: real-world applications from industry'* [48] című könyvben olyan alkalmazási területek találhatóak amelyek az ipar számára is fontosak. Találhatóak benne gazdasági, egészségügyi, oktatási vagy akár biztonsági alkalmazások is. Az alkalmazások többsége szerver oldali megvalósítások, amelyeket böngészőn keresztül lehet elérni, de vannak olyanok amelyek mobil eszközök számára készültek. Az első és legismertebb a DBpedia Mobile [56], amely egy hely alapú szemantikus alkalmazás. Ez azt jelenti, hogy felhasználja a koordinátánkat arra, hogy közelünkben lévő érdekességeket ajánljon nekünk. Ezeket egy térképen jeleníti meg különböző ikonokkal a YAGO kategóriák [54] alapján. Ha ráklikkelünk egy ikonra, akkor az alkalmazás a DBpedia-ról [46] származó leírást és a Flickr-ről[3] származó képeket mutatja nekünk. Ez a megoldás hasonlít az általunk bemutatott alkalmazásokra is: a szemantikus információs rendszerben és a szemantikus böngészőben szemantikus

[3]http://www.flickr.com

adatokat jelenítünk meg a felhasználónak, képekkel, URL-ekkel. A különbség a szűrési feltételeknél van, hisz a DBpedia Mobile korlátozott számú szűréssel rendelkezik, és a komplex szűréshez a felhasználónak kell megírnia a SPARQL lekérdezést. Ezzel szemben az információs rendszernél használt szűrési űrlap dinamikus. Minden attribútumra automatikusan generálódik egy szűrési mező. Másik különbség a használt adathalmazban van. A DBpedia Mobile egy adott tudásbázisra lett kifejlesztve, ezzel szemben az általunk bemutatott alkalmazások bármilyen adathalmazon képesek működni. A szemantikus információs rendszer alkalmazás bármilyen cégadatot képes megjeleníteni. A szemantikus böngésző pedig több SPARQL végpontot alkalmaz az adatok elérésére. Az mSpace mobile [39] volt első olyan rendszer, amely biztosította a szemantikus webet a mobilok számára. Ennek a rendszernek is kliens-szerver felépítése van, úgy hogy több adatbázishoz képes csatlakozni egy loadbalancer (terhelés elosztó) segítségével. Szintén hasonló az OntoWiki Mobile [81], amely egy olyan mobilra készített alkalmazás, ami az OntoWiki keretrendszert [69] használja forrásnak és segítségével saját szemantikus webes alkalmazásokat készíthetünk.

Mind a két rendszernek célja, hogy szemantikus adatokat jelenítsük meg a felhasználó számára. A legtöbb rendszer úgy működik, hogy megadjuk milyen adattagot hogyan kell megjeleníteni. Erre ad egy leírást a Fresnel [43]. Itt definiálhatunk szóhalmazokat, amelyek lehetőséget adnak a felhasználóknak, hogy leírják az adataikat. Egy ilyen szóhalmaz két részből tevődik össze: lencsék (lenses) és formátumok (formats). A lencsék leírják az adatnak azon részét amelyet a felhasználó látni szeretne, a formátumban pedig megadjuk, hogy a különböző adatokat hogyan szeretnénk megjeleníteni. A lencséket egyszerű szűrő feltételekkel tudjuk megfogalmazni, amelyek olyan tulajdonságot vizsgálnak, mint az adott objektum típusa. Ha erősebb feltételeket szeretnénk megfogalmazni, akkor használhatjuk a Fresnel saját nyelvét az FSL-t (Fresnel Selector Language) vagy írhatunk SPARQL lekérdezéseket. Ez a megközelítés használatos a mi információs rendszerünknél is, ahol egy konfigurációs fájlban adjuk meg, hogy mely adattagot szeretnék megjeleníteni és hogy mi annak a típusa. A típus alapján a megjelenítés már automatikus.

Berners-Lee és társai bemutattak egy RDF böngésző és szerkesztő eszközt, amit Tabulatornak [40] neveztek, mely lehetőséget ad arra, hogy egy RDF erőforrásról indulva végig navigáljunk a szemantikus gráfon. Az alkalmazásban lehetőségünk van váltani a táblázatos, térképes vagy idővonalas nézet között.

Bizer és Gauß elkészítette a Disco Hyperdata Browser-t [47], ami egy egyszerű navigációt biztosít a szemantikus weben. A böngésző minden adatot megjelenít egy HTML oldalon, ami az adott erőforráshoz tartozik. Az erőforrásokra kattintva természetesen navigálha-

tunk új oldalakra. A böngésző dinamikusan gyűjti össze az adatokat a navigáció közben, hasonlóan a mi szemantikus böngésző megoldásunkhoz.
Micsik és társai szintén készítettek egy szemantikus böngészőt a LODMilla-t [105], ami lehetőséget ad arra, hogy navigálni tudjunk a LOD felhőn. Az alkalmazás célja az RDF gráfok vizuális megjelenítése egy weboldalon keresztül. Első lépésként le kell kérdeznünk egy erőforrást, amit megjelenítünk a felületen, ezután ennek az erőforrásnak a tulajdonságait le tudjuk kérdezni és ki tudjuk választani azokat, amiket meg szeretnénk jeleníteni. Az alkalmazás előnye ezenkívül, hogy a megjelenített gráfot el tudjuk menteni és meg tudjuk osztani másokkal.

3.2.2. Beltéri navigáció

A beltéri navigációra több próbálkozás is volt, melyet jól összefoglaltak Huang és társai [70]. A navigációhoz használhatunk jeleket (WiFi, szenzorok, bluetooth, infravörös), térképeket (2D, 3D), szöveges információkat vagy környezet függő eszközöket (kiterjesztett valóság). Baus, Krüger és Wahlster [28] egy olyan megoldást mutattak be, ahol különböző technikák közös alkalmazásával határozták meg a felhasználó pozícióját. Amikor a cikk készült a mobil eszközök még nem voltak annyira fejlettek, hogy alkalmazni tudják ezt a rendszert. A cikkben is még PDA-kat használtak.
Mulloni és társai [72] egy valós idejű marker alapú eszközt készítettek mobil készülékekre. A térkép alapú megoldásukat összevetették GPS alapú megvalósítással. Az eszközük segítségével elkészítettek egy beltéri navigációs rendszert, amely egy konferencián be is lett mutatva. Hasonlóan a mi rendszerünkhöz ők is térképet és markereket használtak, amelyek megmutatták a helyes irányt. Az előnye a mi rendszerünknek, hogy nincs szükség mindig újratelepíteni az alkalmazást, ha új térképre van szükségünk. A rendszerünk egy QR kód leolvasásával letölti az új térképet és már használható is. A Mulloni-félében le kell tölteni a szoftver új verzióját, ha a térkép változik. Egy másik cikkükben [88] bemutattak egy kiterjesztett valóság interfacet beltéri navigációkhoz. Ebben navigációs utasításokat adtak információs pontok alapján. Az általunk fejlesztett rendszerben a markereken megjelenő nyilak hasonlóan viselkednek.

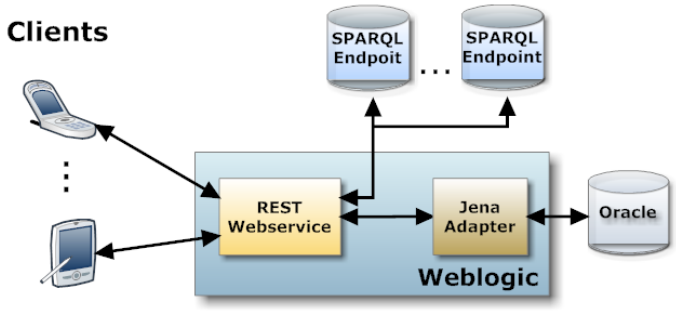
3.3. Szemantikus információs rendszer [2]

3.3.1. Információs rendszerek

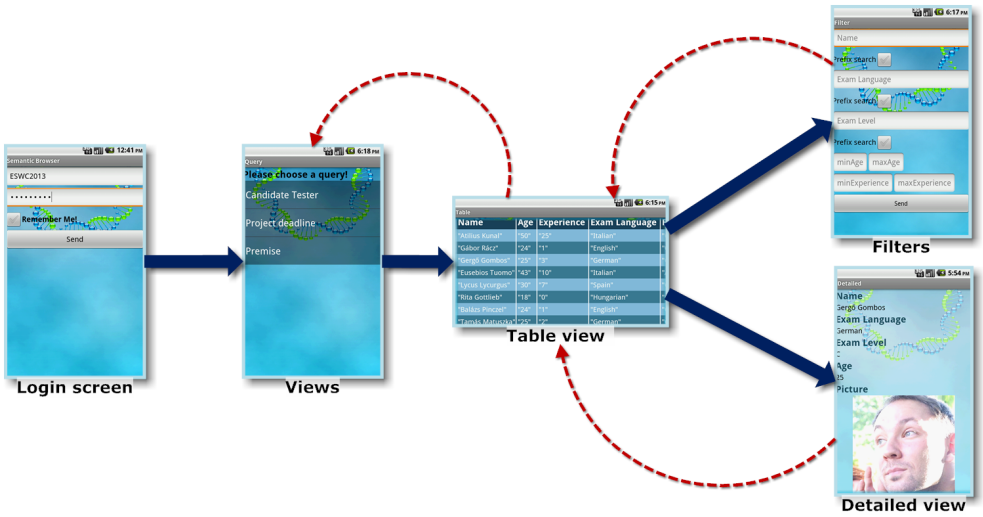
Egy szemantikus információs rendszer elkészítésénél több kihívásra kellett megoldást találnunk. Az első probléma a szemantikus adatok strukturálatlansága, ami miatt az adatokat nem jeleníthettük meg statikus űrlapokon, hanem az adatok alapján dinamikusan kellett létrehozni őket. Például, ha az adat egy URI, akkor linkelhetőnek kellett lennie, ha egy kép, akkor meg kellett jeleníteni. A dinamikus megjelenítés segítségével bármilyen szemantikus adat megjelenítésére alkalmassá vált a rendszerünk. A másik probléma a hozzáférési jogok megfogalmazása, hisz egy vállalati rendszernél fontos, hogy ki milyen adatokat láthat. Mivel az oldalak dinamikusan jönnek létre, így egy olyan struktúrát kellett elkészítenünk, ahol ugyanazt az adatforrást használva, esetlegesen kevesebb információt kaphatnak bizonyos felhasználók. Emiatt a hozzáférési jogokat szemantikus adatként tároltuk. Használati szempontból meg kellett oldani, hogy a felhasználó elől elrejtsük a SPARQL lekérdezéseket. A legtöbb felhasználó jellemzően nem ismeri a szemantikus webet és a SPARQL-t. Erre megoldásként dinamikus űrlapokat készítettünk, amelyeket a felhasználók már képesek voltak használni. Ezen űrlapokba írt értékekből készült végül a végső SPARQL lekérdezés. Egy lekérdezés futtatásánál megoldást kellett találnunk arra, hogy ne csak a vállalat adatain, hanem külső publikus végpontokról is lehessen adatokat lekérdezni. Ahogy a 3.1. ábrán látjuk a rendszer a kliens-szerver modellt követi. Minden művelet a szerveren történik, így bármilyen kliens képes rá csatlakozni, akár olyanok is, amelyek korlátozott CPU-val vagy memóriával rendelkeznek, mint például a mobil készülékek vagy tabletek. A kliensnek semmi más feladata nincs, mint megfogalmazni a lekérdezéseket, majd a szervertől kapott választ megjeleníteni. A kliens bármilyen eszköz lehet, de a prototípust Android készülékre készítettük el. Azért készítettünk mobil alkalmazást, mert a mobilkészülékek általában mindig kéznél vannak és rendelkeznek Internet kapcsolattal, így a felhasználók bármikor tudják használni.

3.3.2. Az alkalmazás

A 3.2. ábra mutatja a rendszer folyamatát és működését. A belépés után a felhasználó egy előre definiált nézetlistából választhatja ki a számára szükségeset. Ezután következik a szemantikus adatok megjelenítése táblázatos formátumban. Mivel az eredmény több száz vagy ezer sor is lehet, emiatt ez a táblázat dinamikusan töltődik le a szerverről. A szerver lefuttatja a SPARQL lekérdezést és az eredményt egy átmeneti táblába tárolja, ahonnan részekben küldi el a kliensnek. Új adatok lekérdezése akkor történik, amikor



3.1. ábra. Szemantikus Inf. Rendszer: a rendszer felépítése: A rendszer egy REST hívásokon keresztül elérhető webszerver. Ez csatlakozik az interneten tlaálható SPARQL végpontokra, és Jena Adapter [52] segítségével Orcle adatbázisba, amely szemantikus adatokat képes tárolni.



3.2. ábra. Szemantikus Inf. Rendszer: kliens képernyők: a szemantikus információs rendszer telefonon megjelenő képernyői: belépő oldal, táblák listája, táblázatos nézet, szűrési oldal, részletes nézet

a felhasználó a táblázat végére érkezett. A felhasználónak lehetősége van szűréseket is megfogalmaznia, amik a szemantikus adatok típusai alapján jönnek létre. Szöveges adatokra teljes vagy rész tartalmazást, dátumoknál pedig intervallumot lehet szűrni. Kép és dokumentumnál pedig lehet szűrni arra, hogy rendelkezik-e vele az adott objektum. A táblázatos nézetből végül át tudunk térni a részletes nézetre, ahol egy újabb lekérdezés segítségével kinyerjük, az adott objektum adatait.

Konfigurációs állomány

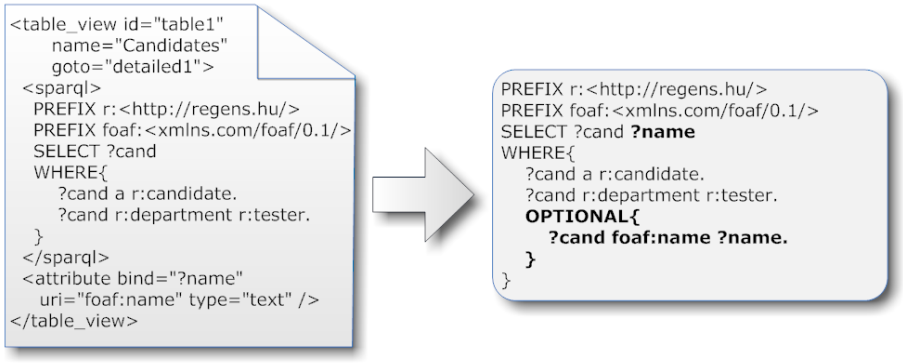
A 6. példa bemutat egy konfigurációs állományt, amelyből a lekérdezéseket készítjük. Ez a példa a Bor ontológiát [27] használja adathalmaznak. Láthatjuk, hogy a *table_view* tagek között fogalmazhatunk meg nézeteket, amelyek közül a felhasználók választhatnak. Egy nézetben mindig meg kell adni, hogy az adott lekérdezés egy lokális adathalmazban, vagy pedig valamely végponton érhető el. Ebben az esetben láthatjuk, hogy a lekérdezést a lokális adatbázisban kell lefuttatni (3. sor). Ezután meg kell adnunk egy alap SPARQL lekérdezést, ami megadja az objektumokat, amelyeknek le lehet kérdezni a további tulajdonságait. Láthatjuk, hogy itt a lekérdezés a vörös és fehér borokat adja vissza. Végül megadjuk azt, hogy mely változó tartalmazza az objektumokat és hogy milyen részletes nézet tartozik hozzá. Ebben az esetben a *?wine* változóhoz a *detailed1* nézet tartozik. A részletes nézetben megadhatjuk, milyen információkat jelenítsünk meg ezekről az objektumokról. Láthatjuk azt, hogy megjelenítjük például a borászatot (winery) is.

Példa 6 Bor ontológia konfigurációs fájl

```
1: <configuration>
2:     <table_view id="wine" name="Red or White Wine" url="wine Wine">
3:         <endpoint>local</endpoint>
4:         <sparql>
5:         PREFIX wine http //www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
6:         PREFIX rdf  <http //www.w3.org/1999/02/22-rdf-syntax-ns#>
7:         SELECT DISTINCT ?wine WHERE
8:         {
9:         ?wine rdf:type wine:Wine .
10:        ?wine wine:hasColor ?color .
11:        FILTER ((?color = wine:Red) || (?color = wine:White)) .
12:        }
13:        </sparql>
14:        <entity bind="?wine" name="Wine" goto="detailed1"/>
15:        <property bind="?body" name="Body" url="wine:hasBody" type="enum"/>
16:        <property bind="?sugar" name="Sugar" url="wine:hasSugar" type="enum"/>
17:        <property bind="?color" name="Color" url="wine:hasColor" type="enum"/>
18:        <property bind="?gra" name="Grape" url="wine:madeFromGrape" type="list"/>
19:    </table_view>
20:    ...
21:    <detailed_view id="detailed1" name="Wine description" url="wine:Wine">
22:        <identity bind="?wine" name="Wine" />
23:        <property bind="?body" name="Body" url="wine:hasBody" type="text" />
24:        <property bind="?sugar" name="Sugar" url="wine:hasSugar" type="text" />
25:        <property bind="?winery" name="Winery" url="wine:hasMarker" type="text"/>
26:    </detailed_view>
27:    </configuration>
```

Lekérdezés generálás

A táblázatos nézetnél található alap SPARQL lekérdezéshez hozzávesszük a további attribútumokat. Minden attribútum rendelkezik egy URI-val, egy névvel és egy típussal. A szerver ezek alapján készíti el a lekérdezéseket: Az attribútumok nevei változókként lesznek hozzáfűzve a lekérdezéshez a SELECT kulcsszó után. A WHERE kulcsszó tartalmazni fogja a megfelelő hármasmintát az attribútum deklarációja alapján egy OPTIONAL kulcsszóval. (OPTIONAL {?objektum <URI><attribútum neve>. }) Ha az adott attribútumra szűrési feltétel van, akkor az OPTIONAL részen belül további hármasminták lesznek. A 3.3 ábra egy kis példán mutatja be a folyamatot. Bal oldalon láthatjuk a konfigurációs fájl egy részét, ami a *Candidates* nézetet írja le. A definíció tartalmazza az alap SPARQL lekérdezést és egy attribútumot, aminek az oszlopneve *name*. Amikor a táblázatos nézetet készíti a szerver, akkor a jobb oldalon szereplő lekérdezés generálódik és fut le.



3.3. ábra. Szemantikus Inf. Rendszer: SPARQL lekérdezés készítés: Baloldalon egy konfigurációs részletet láthatunk, ami alapján elkészítjük a megfelelő SAPRQL lekérdezést. A konfigurációban található egy alap lekérdezés, valamint attribútumok, amelyek OPTIONAL kulcsszóval kerülnek bele a lekérdezés WHERE feltételébe.

Részletes nézetnél ugyanazt az alap lekérdezést alkalmazzuk, mint a táblázatos nézetnél, csak a további attribútumokat a részletes nézet leírásából vesszük.

3.4. Beltéri navigáció szemantikus web és kiterjesztett valóság támogatással [3]

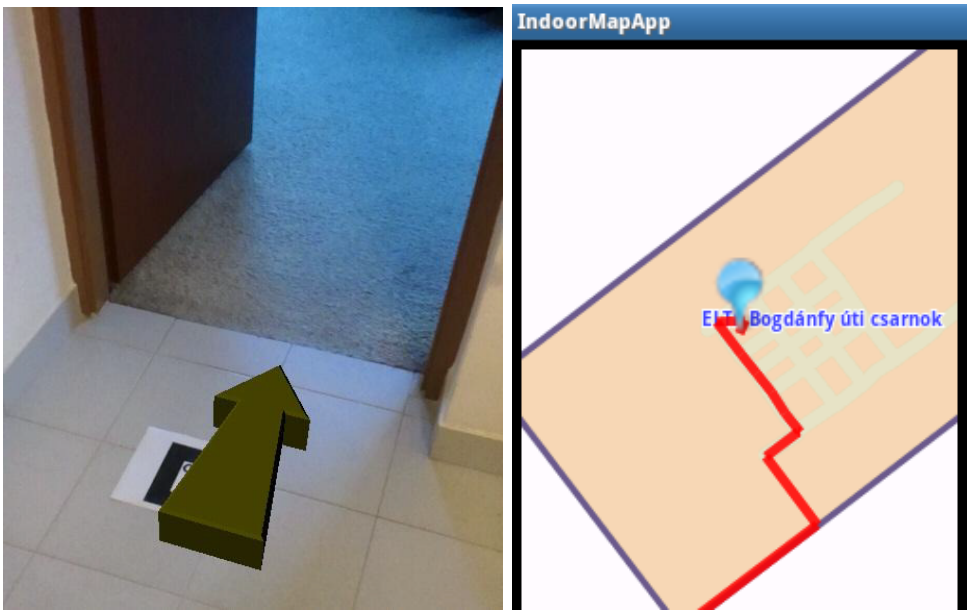
3.4.1. Beltéri navigációs technikák

Az itt bemutatott rendszer motivációja az volt, hogy olyan rendszert készítsünk, amely a már meglévő eszközök alapján képes beltéri navigációra. Az ötletünk azon alapszik, hogy az épületekben elhelyezünk QR kódokat, amelyeket a navigációhoz használunk. A legtöbb publikáció a WiFi alapú navigációt [50, 84, 100] alkalmazza, amely a WiFi jel erőssége alapján határozza meg a felhasználó helyzetét. Megjelölnek a térben különböző úgynevezett *fingerprinteket*, ahol megmérik a WiFi jelek erősségeit. Ezután ha az eszköz érzékeli a jeleket, akkor a legjobban hasonlító mintához fogja a pozíciót meghatározni. A WiFi jelek viszont könnyen elnyelődnek vagy interferálnak egy épületen belül, így ezt kiegészítették irányokkal [84], hisz az elnyelődést okozhatja akár a felhasználó maga is. Nem mindegy, hogy egy WiFi routerral szemben vagy háttal áll. A probléma ezzel a technikával, amit több cikkben is említenek, hogy több routerre van szükség a helymeghatározáshoz. A célunk viszont az volt, hogy ne használjunk több eszközt, mint ami a rendelkezésre áll, emiatt ezt a technikát nem használtuk. A következő módszer, amivel foglalkoztunk a lépésszámláló [86], ami a mobil eszközökben található gyorsulásmérő és iránytű segítségével határozza meg, hogy a felhasználó merre is járhat a térképen. Ez a megoldás használható volt rövidtávú navigációhoz, de hosszabb távon ez is pontatlanul működött. Végül a kiterjesztett valóság (AR) alapú navigációt alkalmaztuk, amely egy kamera által készített képre helyezett nyíllal mutatta meg a helyes irányt.

3.4.2. Az alkalmazás

A rendszer felépítése itt is a kliens-szerver modellen alapszik, ahol a szerver tárolja az épületek térképét és a hozzájuk tartozó szemantikus adatokat. A kliens egy QR kód segítségével le tudja tölteni ezeket az adatokat, valamint a QR kód alapján beállítja a jelenlegi helyzetét. A letöltött szemantikus adat a pozíciók közötti távolságot tartalmazza, amelyet a szerver számolt ki. Ezután a kliens kiválaszt egy adott célpontot amelyhez a Dijkstra algoritmus [18] segítségével kiszámolj a legrövidebb utat, majd indulhat a navigáció. Az alkalmazás kétféle navigációs megoldást alkalmaz, az első egy lépésszámláló[4] segítségével történik, ahol a térképen felrajzoljuk a legrövidebb utat, illetve a felhasználó pozícióját. A másik megoldás pedig a kiterjesztett valóságot [94] alkalmazza, amely meg-

[4]https://github.com/bagilevi/android-pedometer



3.4. ábra. Beltéri navigáció képernyők: Útvonal megjelenítése virtuális valóság (bal) és lépésszámláló segítségével (jobb).

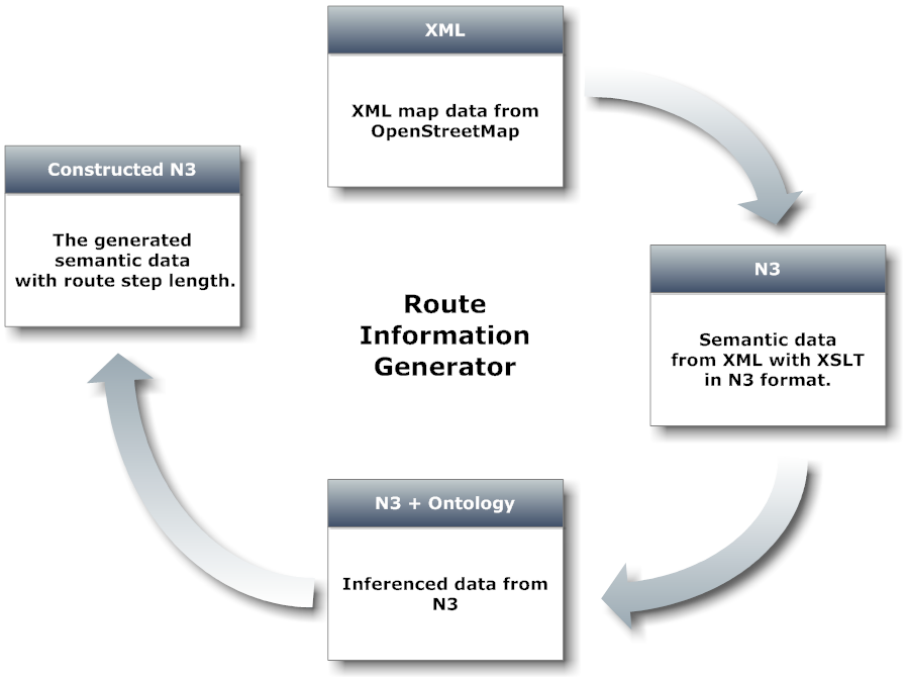
adott pozíciókon lévő markerekhez ad plusz információt. A kétféle megjelenítést láthatjuk a 3.4 ábrán.

A szemantikus adatokkal ellátott térkép generálást mutatja be a 3.5. ábra. Első lépésként elkészítjük az épület tervrajzát az OpenStreetMap térképkészítőjével [61], amely XML formátumban adja meg nekünk a térképet. Ebből készítünk N3 formátumot és szemantikus adatot az INO (Indoor Navigation Ontology) ontologia [33] segítségével. Ezt az adathalmazt kiegészítjük egy új tulajdonsággal a *hasAvailable*-lel, amely megadja mely csomópont érhető el az adott csomópontból, és szimmetrikusként hozzuk létre, majd egy logikai következtető segítségével kiszámoljuk ezeket az új értékeket. Már csak az adott pontok távolságára volt szükségünk, amelyet egy CONSTRUCT lekérdezés segítségével készítettünk el. Ezt a lekérdezést láthatjuk a 7. példán. Ennek eredménye egy olyan szemantikus adathalmaz, amely tartalmazza az épület fontosabb pontjait és a köztük lévő távolságot.

3.5. Szemantikus böngésző mobilra [6]

3.5.1. Szemantikus böngészők

A szemantikus böngészők arra szolgálnak, hogy meg tudjuk tekinteni a szemantikus adatokat. Az RDF alapja, hogy minden erőforrás, objektum egy URI-val legyen



3.5. ábra. Beltéri navigáció útvonal generálás: 1- openstreet map alapján elkészítjük a helyiség térképét. 2- XSLT segítségével N3 formátumot készítünk a térképből. 3- A szemantikus adatokból egy következtető segítségével kiszámítjuk az elérhető pontokat és azok távolságát. 4- Az elkészült adathalmaz alapján elkészítjük a navigáláshoz szükséges útvonal információit.

tárolva. Ez az URI lényegében egy URL, amit ha megnyitunk, akkor egy szemantikus böngészőre kerülünk. A szemantikus böngésző megjeleníti ennek az erőforrásnak az adatait táblázatos vagy gráfos formában. A 3.6. ábrán láthatjuk, ahogy a Budapest erőforrásról (http://dbpedia.org/resource/Budapest) megjeleníti táblázatos formában a DBpedia [56] az állításokat. Láthatjuk, hogy a táblázat két oszlopból épül fel. Az első az állítmányokat, a második pedig a tárgy erőforrásokat tartalmazza. Ha ráklikkelünk egy erőforrásra, akkor a böngésző annak az erőforrásnak a tulajdonságait jeleníti meg. A 3.7. ábrán a SZTAKI által fejlesztett LODmilla böngészőt [105] láthatjuk, amely gráfként jeleníti meg az adatokat. Láthatjuk a Budapest erőforrást és három további erőforrást, ami a *wiki:City*, *dbr:CentralHungary* és *dbr:Hungary*. Látszódik, milyen élekkel kapcsolódik a három csúcs a Budapest erőforráshoz. A kutatás célja egy olyan rendszer készítése, amely mobil eszközökön is képes megjeleníteni a szemantikus adatokat.

Példa 7 Construct lekérdezés a beltérinavigációhoz

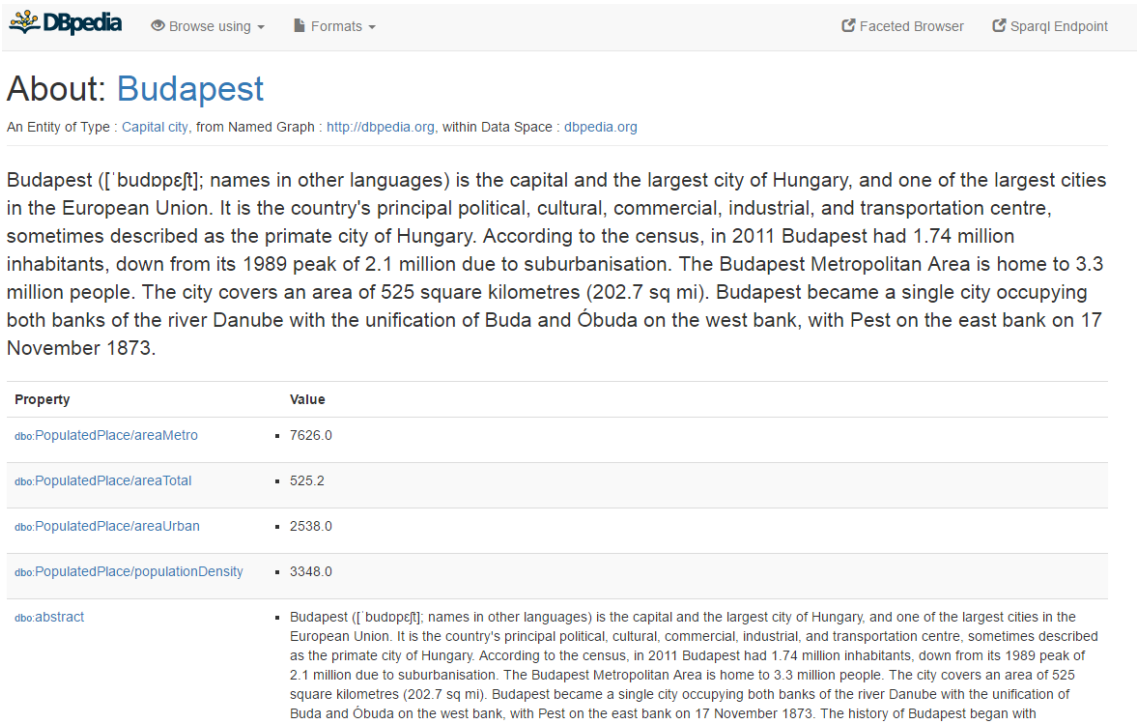
```
PREFIX ex: <http://example.com/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX id: <http://www.indoornavigation.org/indoorontology.owl#>
CONSTRUCT
    ?from id:name ?name .
    ?from rdf:type ?type .
    ?from id:lattitude ?lat1 .
    ?from id:longitude ?lon1 .
    ?from ex:WeightedRoute _:dist .
    _:dist ex:DistanceValue ?distance .
    _:dist ex:DistanceTo ?to .
    ?to id:lattitude ?lat2 .
    ?to id:longitude ?lon2 .
 WHERE
    ?from id:hasAvailable ?to .
    ?from id:name ?name .
    ?from rdf:type ?type .
    ?from id:lattitude ?lat1 .
    ?from id:longitude ?lon1 .
    ?to id:lattitude ?lat2 .
    ?to id:longitude ?lon2 .
    BIND (xsd:float((((?lat1-?lat2)*(?lat1-?lat2)) +
    ((?lon1-?lon2)*(?lon1-?lon2)))) AS ?distance) .

```

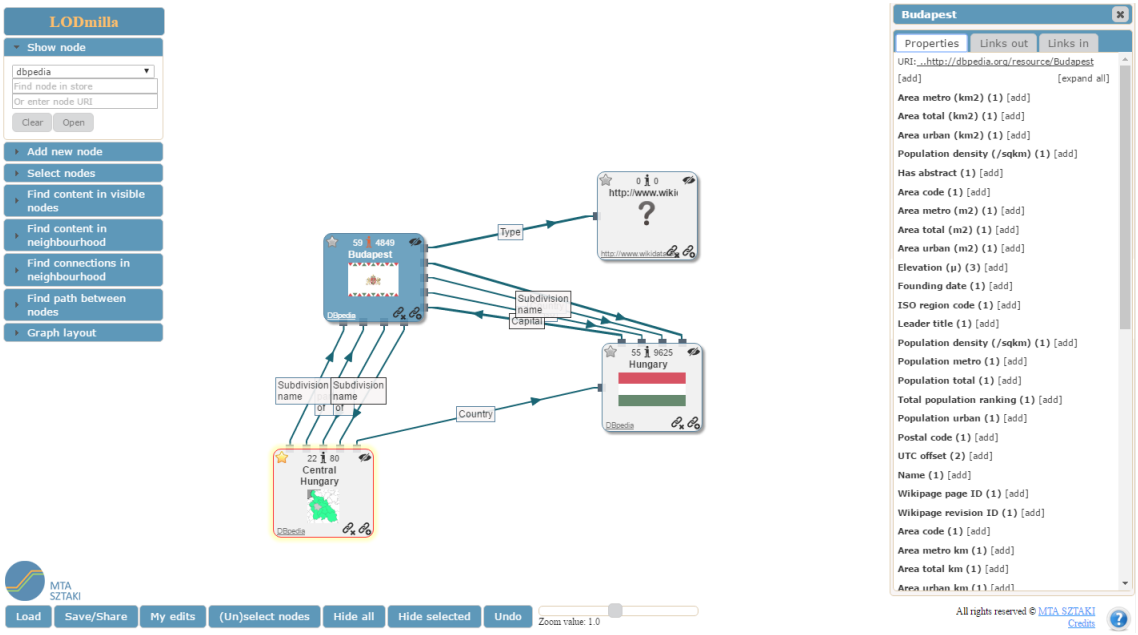
3.5.2. Az alkalmazás

A legtöbb szemantikus böngésző a saját adatbázisában található információkat képes megjeleníteni. A mi rendszerünket viszont úgy készítettük el, hogy az több SPARQL végpontról is képes legyen adatokat lekérdezni. A lekérdezések eredményét ezután elküldte a kliensnek, amelynek csak meg kellett jelenítenie azt. A szemantikus böngészés első lépése egy URI megtalálása. Erre a megoldásra a DBpedia kereső rendszerét[5] alkalmaztuk. A felhasználó egy űrlap segítségével megad egy kifejezést, amire a szerverünk lekérdezi a DBpedia keresőjét és visszaadja a lehetséges URI-kat. A felhasználó ezekből az URI-kból tud elindulni. Az előbb említett űrlapokat láthatjuk a 3.8. ábrán. A kiválasztott URI után a rendszer megjeleníti az adatokat táblázatos formában (3.9. ábra). A táblázatok megjelenítése itt is dinamikusan történik, hogy a kliens oldalon a megjelenítést gyorsítsuk. Láthatjuk az URI-k itt is linkelhetőek, a képek pedig megjelennek. A jobb oldali ábrán láthatjuk, hogy ha olyan adatunk van, amely rendelkezik *geo:lat*, *geo:long* tulajdonsággal, akkor azokat az erőforrásokat térképen is meg tudjuk jeleníteni.

[5] http://dbpedia.org/fct/

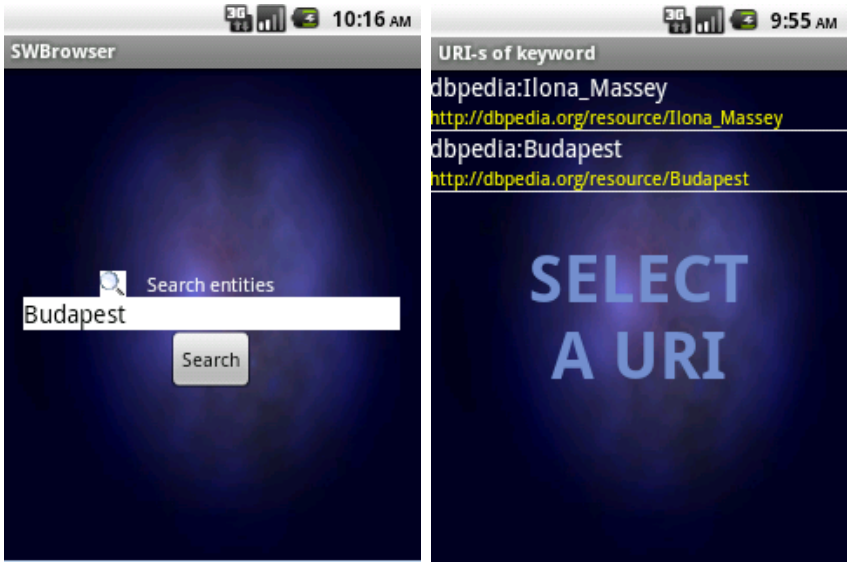


3.6. ábra. A DBpedia szemantikus böngészője



3.7. ábra. A LODmilla szemantikus böngésző

A szerver oldalon fix SPARQL végpontok vannak beállítva, amelyeket a szerver használ a lekérdezések megválaszolására. A végpontok között 8 publikus (dbpedia, factforge, linkedmdb, rkbexplorer, dblp, data.gov, factbook, openlinksw) és 1 saját telepítésű SPARQL



3.8. ábra. Szemantikus böngésző mobilra keresés képernyő: Bal oldalon kulcsszó alapú keresést láthatunk, jobb oldalt pedig a végpontoktól eredményül kapott IRI-ket.

3.9. ábra. Szemantikus böngésző mobilra táblázatos, térképes képernyő: A kiválasztott IRI megjelnítése táblázatos és térképes formátumban.

végpont és 1 publikus (dbpedia) és 2 saját telepítésű FCT (faceted) található. Az FCT keresés a szavak alapján történő URI megtalálásra szolgál. A szerver ezenkívül tartalmazza a gyakran használt PREFIX-eket, amelyeket a prefix.cc-ről töltöttem le. A szerver REST hívásokon keresztül érhető el két végponton. A *service/fct* az FCT hívásokat kezeli, még a *service/browse* a böngészésért felel. A szerver a lekérdezéseket párhuzamosan végzi el az egyes végpontokon. A kapott paraméterek alapján minden egyes végpont lekérdezés egy külön szálban fog futni. A szálak időlimittel rendelkeznek, így ha egy-két végpont nem válaszol, akkor azok nem lesznek benne az eredményben. Ha a lekérdezés FCT-re vonatkozik, akkor a válaszul kapott eredményt visszaküldi a kliensnek. Ha egy IRI-ről szeretnénk további információkat, akkor minden végpontra a szerver két lekérdezést fut-

tat, ahol az első lekérdezés alakja: *?s ?p <IRI>*, még a másodiké: *<IRI> ?p ?o*. A végpontoktól kapott választ a szerver elemzi és típusokkal látja el az információkat. Ha egy elem kép linket tartalmaz, akkor képként jelöli meg, ha szöveget akkor szövegnek, ha IRI, akkor IRI-nek. Mivel az egyes állítások több végponton is szerepelhetnek, amiatt a klienshez küldés előtt az adatokat leszűkítjük, hogy minden állítás csak egyszer szerepeljen benne. Ezután az IRI-ket prefixesíti, hogy minél kevesebb információt kelljen a kliensnek elküldeni. Ezt a prefixesített verziót fogja megjeleníteni a kliens. Az így kapott prefixesített, típusozott eredményekből egy XML-t készít a szerver, amelyet visszaküld a kliensnek.

3.6. Összefoglalás és kapcsolat a tézisekkel

Ebben a fejezetben olyan kliens-szerver architektúrákat mutattam be, ahol a kliens egy mobil készülék. A mobil készülékek számára a szemantikus web használata nem megoldott, hisz a mobilok erőforrásai korlátozottak. Az 1. tézisem arról szól, hogy egy megfelelő szerver segítségével használhatóvá tudjuk tenni a szemantikus webet mobil készülékek számára is. Az első alkalmazás olyan rendszert mutatott be, ahol ipari környezetben használjuk a szemantikus adatokat. A második megoldás egy beltéri navigáció megvalósítása, ahol az objektumok leírására és az útvonal meghatározására használtuk a szemantikus adatokat. Végül egy olyan rendszert mutattam be, ahol a szemantikus web világban ismert szemantikus böngészőket tettük elérhetővé a mobil készülékek számára.

4. fejezet

Szemantikus adatok federált környezetben

4.1. Bevezető

Ahogy a szemantikus böngészőnél láttuk, a szerver feladata megoldani, hogy több SPARQL végpontról tudjon lekérdezni adatokat. Ezeket a rendszereket hívjuk federált rendszereknek és ebben a fejezetben bemutatom a témakörben elért eredményeimet, amelyek önálló munkáim voltak.

A SPARQL 1.1 megjelenésével maga a SPARQL lehetőséget ad arra, hogy lekérdezzünk több végpontot egy lekérdezésen belül. Ehhez a SERVICE kulcsszót tudjuk alkalmazni. A kulcsszónak az első paramétere a SPARQL végpont URL-je, a második a hármasminták, amelyeket ki akarunk értékelni rajta. Ha össze akarunk kapcsolni két végpontot, akkor az egyiken futtatjuk a lekérdezést, amely meghívja a SERVICE-ben szereplő másik végpontot és összekapcsolja az adatokat. Ez elég nagy terhet jelent, emiatt sok végpont nem is támogatja a SERVICE kulcsszó használatát. Vannak olyan végpontok amelyek a lekérdezés bonyolultságát mérik és ez alapján becsülnek meg egy költséget arra, hogy mennyire fogja a lekérdezés leterhelni a rendszert. Ha ez az érték meghalad egy limitet, akkor nem engedi lefuttatni a lekérdezést. Erre jelentenek megoldást az olyan federált rendszerek, amelyek a SERVICE kulcsszó nélkül értékelik ki a lekérdezést. Ezek először megállapítják az egyes hármasmintákról, hogy melyik végponton kell kiértékelni őket. Majd ezek alapján rész-lekérdezéseket készítenek, lefuttatják a végpontokon és az eredményeket összekapcsolják. A szemantikus federált rendszereknél kétféle megoldást alkalmaznak a végpontok kiválasztására. Az első módszer mindig ellenőrzi, hogy az adott végpont képes-e megválaszolni a lekérdezést. Ezt az ASK SPARQL lekérdezéssel teszi meg, amely

igaz-hamis értékkel tér vissza, annak függvényében, hogy a végpont tartalmaz-e olyan adatot, ami illeszkedik a hármasmintára. Ennek előnye, hogy mindig aktuális információval rendelkezik a végpontokról. A hátránya viszont az, hogy minden végpontot le kell kérdezni minden hármasmintára. A másik módszer a katalógus technika, ahol a federált rendszer egy katalógusban tárol információkat a végpontokról és ez alapján fogja eldönteni, mely végpontot használjuk a kiértékelésnél. Ennél a módszernél az egyik probléma, hogy a katalógust karban kell tartani, a másik pedig, hogy mit tároljunk a katalógusban. A legtöbb ilyen katalógus az állítmányokat tartalmazza az adott végpontról, de lehetne tárolni a névtereket is, hisz az egyes adathalmazok általában egy adott területről tárolnak információkat. Vannak például filmes (imdb), geológiai (geonames), vagy wikipedia (dbpedia) információkat tartalmazó adathalmazok. A fejezet első részében bemutatok egy rendszert, amely ezen információk alapján dönti el, hogy a lekérdezés mely végpontokat használja. A feltevés az, hogy ha minden végpont a saját névterét használja, akkor a lekérdezésben szereplő névterek alapján meg tudjuk határozni, hogy az egyes hármasminták mely végponthoz tartoznak. A rendszer működéséhez emiatt a végpontok URL-jére és az ott elérhető névterekre van szükség. Vannak azonban olyanok, amelyek több végponton is szerepelhetnek, emiatt lehetnek olyan hármasminták, amelyre nem tudunk egyértelműen meghatározni végpontot. Ezeket az eseteket fogom konfliktusos helyzetnek nevezni és bemutatom a feloldásának egy módszerét. A módszer azon alapszik, hogy a hármasmintákat összekapcsoló változók segítségével megnézem, hogy a többi hármasminta mely végponton fog futni.

Egyik ok, ami miatt kevesen használják a szemantikus webet, az a bonyolultsága. Ahhoz, hogy megfelelő lekérdezéseket tudjunk készíteni, ismernünk kell a végpontok elérhetőségét, valamint az ott tárolt adatok struktúráját. Mivel az adatok strukturálatlanok, emiatt nem tudjuk egyszerűen átlátni az adathalmazt. Ha a relációs adatbázisok világára gondolunk, akkor ott egy lekérdezés megírásához meg kell vizsgálnunk a lehetséges táblákat és azok oszlopait, majd ezen információkkal már készíthetünk lekérdezést. A szemantikus web világában viszont ez nem lehetséges. Ha egy relációs adatbázis klienst alkalmazunk a lekérdezések megírására, akkor gépelés közben a kliens felajánlja a lehetséges táblákat, oszlopokat, amely segít nekünk a lekérdezés elkészítésében. Ezt a funkciót ültettem át a szemantikus web világára. A fejezet második része ezen megoldás bemutatásával foglalkozik. A prototípus alkalmazás egy böngészőből használható SPARQL lekérdezés készítő felület, amely ajánlásokat tesz hármasmintákra. Ennek egyik alapja, hogy ha a változó rendelkezik *rdf:type* információval, akkor kinyerjük az adathalmazból, hogy milyen állítmányokkal rendelkeznek ezek az osztályok. A másik alapja az *rdfs:range*, amely meg-

mondja, hogy egy állítmányhoz tartozó tárgynak milyen típusúnak kell lennie. A legtöbb ontológia rendelkezik ilyen információval. Ha tudjuk ezt az információt, akkor az ott található változóra ismét tudunk ajánlásokat tenni a típusa alapján. A rendszer ezenkívül prefixeket is képes ajánlani, hisz a legtöbb névtérnek megvan a megszokott rövidítése, mint pl. a DBpediának *dbpedia* vagy *dbr*. Ha a felhasználó ilyen rövidítést alkalmaz a lekérdezésében, akkor a rendszer felajánlja neki, hogy használja a *PREFIX* kulcsszót a megfelelő értékkel. A mögöttes motor egy federált rendszer, amelybe több végpont került bekonfigurálásra.
Amikor egy ilyen ajánlás elkészül, akkor olyan információk birtokába is kerülünk, amelyeket felhasználhatunk arra, hogy megtaláljuk azokat a végpontokat, amelyek képesek megválaszolni a lekérdezést. Ezután bemutatom, hogy ezen információk birtokában hogyan tudunk javítani a lekérdezések kiértékelésének válaszidején, hisz a felhasználó által választott hármasmintákkal tudjuk, hogy mely végpontokat érdemes használni. Így mind a katalógus technikánál mind pedig az ASK lekérdezéseknél kevesebb végpontot kell ellenőriznünk.

4.2. Kapcsolódó munkák

4.2.1. Federált rendszerek

A SERVICE kulcsszó használatáról a SPARQL 1.1 funkcióit bemutató cikkben [101] találhatunk részletesebb leírást. A cikk bemutatja a lekérdezés szemantikáját, a kiértékelését és az olyan problémás lekérdezéseket ahol a SERVICE kulcsszó paramétere egy változó.

A SPARQL 1.1-et viszont nem minden végpont támogatja. Buil-Aranda és társai [80] a SPARQL végpontokat tesztelték. Megvizsgálták mennyire elérhetőek a végpontok, és hogy mennyire támogatják az új funkciókat, amit a SPARQL 1.1 vezetett be. Verborgh és társai azt is észrevették [118], hogy a SPARQL végpontok elérhetősége és rendelkezésre állása alacsony. Erre a problémára ők egy kliens oldali megoldást adnak. Az ötletük az, hogy nem küldenek bonyolult lekérdezéseket a végpontoknak, így azok könnyebben tudják megválaszolni azokat. Ehhez kapcsolódik Buil-Aranda és társai [111] által észrevett jelenség, hogy nem minden federált megoldás ad megfelelő eredményt. Cikkükben elemezték a federált SPARQL kiértékelési stratégiákat és az okot arra vezetik vissza, hogy a végpontok csak korlátozott számú eredményt képesek visszaadni. Ezt a problémát láthatjuk a névtér alapú végpont kiválasztásos eredményemnél is, amit ebben a fejezetben mutatok be.

A federált rendszerek működéséről egy összefoglalót készítettek Rakhmawati és társai [108]. Bemutatják a federált rendszerek komponenseit és az ismert végpont választási

stratégiákat: ASK lekérdezés és katalógus technika, amiket én is alkalmazok az ajánlórendszeremnél. Ismert eszközök, amelyek ezeket a technikákat alkalmazták: az ADERIS [87], a FedX [91] és a DARQ [45]. Az ADERIS [87] a katalógus technikát alkalmazza és állítmányokat tárol ahhoz, hogy ki tudja választani a megfelelő végpontokat az allekérdezésekhez. A rendszernek van egy beállítási fázisa, amikor az állítmányokat gyűjti össze. A FedX [91] lokális adatbázist és SPARQL végpontokat is képes használni. A végpont választási problémát ASK lekérdezésekkel valósítja meg és rendelkezik lekérdezés optimalizációval is. Egy FedX-en alapuló keretrendszer a FedSearch [106], amelyben lehetőségünk van szöveg alapú keresésre, ami a SPARQL lekérdezéseknél nehézkes. A DARQ [45] (distributed ARQ) a JENA ARQ [52] egy kiegészítése, ami a federált lekérdezésekhez készült. A lekérdezéseket költségalapú lekérdezés optimalizációval állítja elő, a végpont választása pedig állítmány alapú.

A végpont kiválasztásához információkat gyűjtök a végpontokról, emiatt meg kell említenem a LODStats [92] rendszert, ami különböző statisztikákat gyűjt az adathalmazokról. A gyűjtött információkat a kiértékelésnél alkalmazom.

4.2.2. Ajánlórendszerek, vizuális lekérdezés készítők

Hoefler [102], Campinas [112] és Han [120] is arról ír a cikkében, hogy a szemantikus web használata bonyolult, mert nehézkes egy SPARQL lekérdezés megírása. A felhasználók nem ismerik a végpontok URL-jét és nem tudják milyen adatok vannak ott tárolva. Hoefler megfigyelte, hogy a felhasználók könnyebben használják a táblázatos adatszerkesztőket, mint például az Excel-t, emiatt az ötlete az volt, hogy a szemantikus adatokat táblázatos formában jelenítsük meg. Campinas egy olyan adat alapú kiegészítést készített, amely állítmányokat, osztályokat ajánl a felhasználónak. Han megoldása egy grafikus megvalósítás, ahol a felhasználó egy lekérdezés gráfot ad meg és szövegesen annotálja a csúcsokat. Mindhármuk célja az volt, hogy a felhasználók számára egy könnyen használható eszközt készítsen.

Kramer és társai [103] leírták, hogy a szemantikus web lekérdezése nem hasonlít a relációs adatbázisoknál használtakra. Céljuk, hogy a SPARQL-t automatikus kiegészítésekkel tudjuk elkészíteni. Ezt úgy valósították meg, hogy a lekérdezési logokból indexeket építettek a lekérdezésekre és az adathalmazokra is. Ha a felhasználó beír egy '<' szimbólumot, akkor a rendszer felajánlja a lehetséges IRI-ket. Ha egy '?'-et akkor pedig változókat, mely után a korábbi lekérdezések alapján állítmányokat fogja ajánlani. Az ajánlórendszerem ezzel szemben valós időben gyűjti az ajánlásokat. Lehmann és társai [85] is bemutattak egy technikát SPARQL készítésre. Az ő megoldásuk a kérdés-válasz és az aktív tanulás

technikákon alapszik. A felhasználó megad egy lekérdezést, amire a rendszer ajánlásokat tesz. A felhasználó az ajánlásokból kiválaszt egy pozitív példát, ami alapján a rendszer újabb ajánlásokat képes készíteni. Ez az iteráció addig tart, amíg a felhasználó el nem ér a megfelelő lekérdezéshez, vagy nincs további tanulható lekérdezés. Rietveld és társai készítették el a Yasgui rendszert [109], amely egy felhasználóbarát webalapú SPARQL kliens. A rendszer proxyt használ a különböző végpontok elérésére és rendelkezik automatikus kiegészítéssel is, ami prefixeket, névtereket és állítmányokat ajánl több végpontról, de a kiértékeléshez csak egy végpontot használhatunk.

4.3. ASM modell a federált rendszereknek [5]

A szemantikus federált rendszerekhez elkészítettem egy ASM (Abstract State Machine) leírást. Az ASM egy matematikai alapokon épülő keretrendszer rendszerek tervezésére és analizálására [93]. Egy ASM leírás megadja a rendszer felé elvárt követelményeket olyan formában, hogy az mindenki számára olvasható legyen. A használatot inspirálta a Grid Rendszerek ASM modellje [29], amely hasonlóan elosztott, párhuzamos működésűek, mint a federált rendszerek. Egy ASM algebra univerzumokból, függvényekből és szabályokból épül fel. Az univerzumok tartalmazzák a konkrét elemeket, a függvények az univerzumok közötti kapcsolatot biztosítják, a szabályok pedig feltételes lefutást tesznek lehetővé. Minden szabály tranzakciós egységekként fut le. Az ASM modell egy általános alap modellből és annak finomításaiból épül fel. Az alábbiakban leírom a federált rendszerek felé elvárt funkciókat ASM modell segítségével, majd finomítom azt a SPARQL ajánló rendszerek és a szemantikus böngészők szempontjából.

4.3.1. Modell a federált rendszerekhez

Egy szemantikus federált rendszer (FEDERATED univerzum) működése a következő. A rendszer fogad egy lekérdezést (QUERY univerzum), amelyet megfuttat több SPARQL végponton (ENDPOINT univerzum) és az eredményeket összegezve visszaadja a lekérdezésünk végső eredményét (RESULT univerzum). Ahhoz, hogy a rendszer leírása elég általános legyen, az alap modellben nem foglalkozunk azzal, hogy mi alapján választja ki a végpontokat. Annak működését a finomított modellben adom meg. Az univerzum értékek között a $true, false, undef$ értékek alapból megtalálhatóak, ezeket nem kell definiálni.

A rendszer leírásához szükségünk van egy $fstate : FEDERATED \rightarrow \{wait, start\_req, running\}$ függvényre, amely megadja a federált rendszer állapotát. Ez az állapot lehet $wait$, ha a rendszer várakozik a kérésre, lehet $start\_req$, amikor a végpont

hívásokat készíti elő, és *running*, ha futtatja a lekérdezéseket. Ha egy kérés érkezik a rendszerbe, akkor azt össze tudjuk kapcsolni egy $fworkingOn : FEDERATED \rightarrow QUERY$ függvénnyel, amely azt mondja meg, hogy a federált rendszer milyen lekérdezésen dolgozik. Egy federált rendszer a lekérdezéseket kérésekké alakítja. Az alap modellben nem adjuk meg, hogy a kérés pontosan micsoda. Ez mindig az adott federált rendszertől függ. A lekérdezés és a kérések összekapcsolását a $reqQuery : REQUEST \rightarrow QUERY$ függvénnyel tudjuk megadni.

Egy kérés mindig egy adott végponton fog futni. A végpont és a kérés összekapcsolására az $eworkingOn : ENDPOINT \rightarrow REQUEST$ függvény ad lehetőséget. Egy kérés indítása függ a végpont állapotától. Kezdetben a végpontok *waiting* állapotban vannak. Ezekre a végpontokra lehet küldeni a kéréseket. Ahhoz, hogy tudjuk az állapotokat, szükségünk van egy $estate : ENDPOINT \rightarrow \{running, waiting, finished\}$ függvényre. A végpontok állapota akkor változik meg, ha egy esemény bekövetkezik. Egy esemény bekövetkezését szintén egy függvénnyel tudjuk leírni: $event : ENDPOINT \rightarrow \{timout, finish\}$. A *timeout* esemény akkor következik be, ha a végpontnak kiküldött kérés túl sokáig fut. Ez szükséges, hogy a rendszer mindenképp adjon választ. A $finish$ esemény akkor következik be, ha a végpontnak sikerült a lekérdezést megfuttatnia. Az eredményeket a $rres : REQUEST \rightarrow RESULT$ függvénnyel tudjuk visszanyerni. A federált rendszer ezeket az eredményeket összesíti, amelyre az alap modellben szintén nem térünk ki, hogy hogyan. Végül a végleges eredménye egy lekérdezésnek a $qres : QUERY \rightarrow RESULT$ függvénnyel érhető el.

A modell működéséhez szükséges egy inicializációs lépés. Itt a modell minden elemét alaphelyzetbe állítjuk. Először a végpontokat állítjuk alaphelyzetbe: $\forall e \in ENDPOINT : estate(e) := waiting; eworkingOn(e) := undef$. Majd a federált rendszer (f) alapállapotát is beállítjuk: $fstate(f) := wait$.

Most leírjuk a rendszer működését a Szabályok (Rules) megadásával.

Szabály 1 (Lekérdezés küldése a federált rendszernek)

Az első szabály megadja, hogy mi történjen, ha a rendszer kap egy lekérdezést ($q \in QUERY$). A lekérdezés csak akkor tud futni, ha a rendszer épp lekérdezésre vár. Ekkor a federált rendszer állapota átvált az előkészítés állapotba, valamint rögzítjük, hogy a rendszer ezen a lekérdezésen fog dolgozni és ennek nincs még eredménye.

Algoritmus 8 Szabály 1 (Lekérdezés küldése a federált rendszernek)

```
1: if fstate(f) = wait then
2:     fworkingOn(f) := q
3:     fstate(f) := start_req
4:     qres(q) := undef
5: end if
```

Szabály 2 (A federált rendszer elküldi a lekérdezéseket a végpontokhoz)

A lekérdezések kiértékeléséhez szükség van a kérések készítésére. A következő szabály, azt adja meg, hogy minden olyan végpontra, amely *waiting* állapotban van, készítünk egy kérést. Jelen esetben nem foglalkozunk azzal, hogy a kérés az mi is valójában. Egyes rendszereknél ez lehet az egész lekérdezés, más rendszerekben ez csak egy feltétel a lekérdezésből. Egy kérés készítésénél beállítjuk, hogy a kérés melyik lekérdezésen dolgozik, valamint hogy melyik végponton fut. A végpont állapotát átállítjuk futó állapotra, a kérés eredményét pedig $undef$-re.

Algoritmus 9 Szabály 2 (A federált rendszer elküldi a lekérdezéseket a végpontokhoz)

```
1: if fstate(f) = start_req && fworkingOn(f) = q then
2:     for all  e ∈ ENDPOINT do
3:         if estate(e) = waiting then
4:             EXTEND REQUEST by req with
5:                 reqQuery(req) := q
6:                 eworkingOn(e) := req
7:                 estate(e) := running
8:                 rres(req) := undef
9:             end EXTEND
10:        end if
11:        fstate(f) = running
12:    end for
13: end if
```

Egy szabályban az EXTEND jelenti, hogy új elemet hozunk létre az adott univerzumba (itt: $REQUEST$).

Szabály 3 (Végpont állapota megváltozik)

Egy kérés két állapotban érhet véget. Az egyik, hogy a lekérdezés minden gond nélkül lefutott. A második állapot pedig, ha nem tudott lefutni egy bizonyos időn belül. Mindkét esetben egy *event* keletkezik. A kérés eredményét hozzávesszük a lekérdezés eredményéhez a '+' operátorral. Itt megint nem foglalkozunk azzal, hogy ez az operátor hogy működik vagy mit csinál.

Algoritmus 10 Szabály 3 (Végpont állapota megváltozik)

```
1: let req = eworkingOn(e)
2: if event(e) = finish || event(e) = timeout then
3:     eworkingOn(e) = undef
4:     estate(e) = finished
5:     qres(q) := qres(q) + rres(req)
6:     rres(req) := undef
7:     REQUEST(req) = undef
8: end if
```

A $REQUEST(req) = undef$ jelentése, hogy az adott elemet (req) kivesszük az univerzumból ($REQUEST$).

Szabály 4 (Terminálás)

Az utolsó folyamat a terminálló folyamat, aminek a feltétele, hogy ha minden végpont állapota átváltott *finished*-re, akkor minden kérés lefutott, és az eredményt megkaptuk a $qres(q)$-ban. Ezután annyi dolgunk van, hogy a rendszert visszaállítsuk a kezdeti állapotba, hogy újabb kéréseket tudjon fogadni.

Algoritmus 11 Szabály 4 (Terminálás)

```
1: if ∀ e ∈ ENDPOINT : estate(e) = finished && fstate(f) = running then
2:     for all e ∈ ENDPOINT do
3:         estate(e) := waiting
4:         fstate(f) := wait
5:         fworkingOn(f) := undef
6:     end for
7: end if
```

4.3.2. Finomított modell SPARQL ajánlórendszerhez [5]

A korábbi modellben arról beszéltünk, hogy egy lekérdezés kerül a rendszerbe és a federált rendszer ezt fogja feldolgozni. Ezt a modellt finomítjuk a jelenlegi feladatunkhoz. A feladat arról szól, hogy lekérdezéseket szeretnénk készíteni, így a bemenete a rendszernek nem egy QUERY, hanem annak csak egy része. Emiatt bevezetünk új univerzumokat. A feladatunkban két feladatra koncentrálunk. Az egyik, hogy a prefixeket meg tudjuk határozni. Ehhez szükség van a SHORTPREFIX és a LONGPREFIX univerzumra, amik a prefixek rövid és teljes változatát fogják tárolni. A második cél amit meghatároztunk, hogy az ismert hármasmintákból új feltételeket tudjunk megadni. Ehhez bevezetjük a CONDITION univerzumot.

A finomított modellnek új elvárásokat fogalmazunk meg. A prefixeket meg kell tudnunk határozni anélkül, hogy terhelnénk a végpontokat. Ez amiatt lehet, mert a prefixek jellemzően fixek, így ezt konstansként tudjuk kezelni. A REQUEST-ek függnek a CONDITION-től, és csak akkor küldünk lekérdezést a végpontnak, ha típus van a feltételben.

Az új univerzumokhoz új függvényekre is szükség van. Az első a $prefMapped : SHORTPREFIX \rightarrow LONGPREFIX$, ami a prefixek mappelését adja vissza nekünk. A lekérdezésekben a rövid prefixek feloldására van szükségünk, így ez a mappeles egyirányú. Mivel a QUERY-t most több részre osztottuk fel, így szükség van a $pbelongsTo : SHORTPREFIX \rightarrow QUERY$ és a $cbelongsTo : CONDITION \rightarrow QUERY$ megfeleltetésekre. Feltételeket úgy tudunk ajánlani, ha van olyan feltétel, amelynek van típusa (*rdf:type*). A típus ellenőrzésére bevezetjük a $hasType : CONDITION \rightarrow \{true, false\}$ függvényt. Ahhoz, hogy a lekérdezés egyáltalán tartalmaz-e olyan részeket, amelyekkel mi foglalkozni akarunk a $hasCondition : QUERY \rightarrow \{true, false\}$ és a $hasPrefix : QUERY \rightarrow \{true, false\}$ függvények fogják megadni. Egy korábbi függvényen is változtatnunk kell, mert a *reqQuery*-nek eddig a QUERY volt a tartománya, melyet átalakítunk CONDITION-re.

Az inicializáló lépés annyiban változik, hogy a $prefMapped$ függvényt feltöltjük a prefixek rövid és hosszú reprezentációjával. A pontos megvalósítást a modell nem tartalmazza. A másik lépés, hogy a rendszerbe kerülő (al-)lekérdezés alapján beállítjuk a $cbelongsTo, pbelongsTo$ függvények értékeit.

Szabály 1 (Finomított szabály az ajánlórendszerekhez)

Az első szabályon egy minimális változtatásra van szükség. Az alap modellben a rendszer minden esetben küldte a lekérdezést a végpontoknak. A finomított modellben csak akkor foglalkozunk a lekérdezéssel, ha az tartalmaz PREFIX-t, vagy CONDITION-t.

Algoritmus 12 Szabály 1 (Finomított szabály az ajánlórendszerekhez)

```
1: if fstate(f) = wait && (hasPrefix(q) || hasCondition(q)) then
2:     fworkingOn(f) := q
3:     fstate(f) := start_req
4:     qres(q) := undef
5: end if
```

Szabály 2 (Finomított szabály az ajánlórendszerekhez)

A második változtatás az alap modellhez képest, hogy a rendszer a prefixeket megválaszolja, és a QUERY-k helyett csak a CONDITION-ket fogja elküldeni az endpointoknak.

Algoritmus 13 Szabály 2 (Finomított szabály az ajánlórendszerekhez)

```
1: if fstate(f) = start_req && fworkingOn(f) = q then
2:     if hasCondition(q) then
3:         for all c ∈ CONDITION do
4:             if cbelongsTo(c) = q && hasType(c) then
5:                 for all e ∈ ENDPOINT do
6:                     if estate(e) = waiting then
7:                         EXTEND REQUEST by req with
8:                         reqQuery(req) := c
9:                         eworkingOn(e) := req
10:                        estate(e) := running
11:                        rres(req) := undef
12:                        end extend
13:                    end if
14:                end for
15:            end if
16:        end for
17:        fstate(f) = running
18:    end if
19:    if hasPrefix(q) then
20:        for all p ∈ PREFIX do
21:            if pbelongsTo(p) = q then
22:                qres(q) := qres(q) + prefMapped(p)
23:            end if
24:        end for
25:    end if
26: end if
```

4.3.3. Finomított modell a szemantikus böngészőkhöz [6]

A szemantikus böngésző egy adott URI-ról gyűjt információt. Emiatt a QUERY ebben az esetben URI vagy STRING típusú a finomított modellben. Az URI a kiválasztott entitás, a STRING pedig egy szöveg (vagy kulcsszó) amihez az URI-kat szeretnénk megkapni. Mivel az alapmodell univerzuma nem változott, emiatt nincs szükségünk arra, hogy a függvényeket változtassuk. Másfelől mivel a lekérdezések most STRING vagy URI típusúak, a következő függvényeket kell bevezetnünk: $isIRI : QUERY \rightarrow \{true, false\}$ and $isString : QUERY \rightarrow \{true, false\}$. Ha a lekérdezés egy URI, akkor egy SPARQL végpontra van szükségünk, ha pedig STRING típus a kérés akkor viszont egy erőforrás keresésre van szükség. Ezek a végpontok egy adott szóra URI-kat adnak vissza.

Ilyen végpontot biztosít a DBpedia[1] is. Továbbá arra is szükségünk van, hogy a végpont képes-e megválaszolni a kérésünket. Ehhez a következő függvények definiálására van szükségünk: $canIRI : ENDPOINT \rightarrow \{true, false\}$ and $canString : ENDPOINT \rightarrow \{true, false\}$. Az előbb említett tulajdonságokat az inicializációs fázisban állítjuk be.

Szabály 2 (Finomított szabály a szemantikus böngészőkhöz)

A szabályok közül csak a 2. szabályt kell megváltoztatnunk, mert a finomított rendszer olyan lekérdezéseket futtat, amelyek speciálisak a szemantikus böngészőkhöz. A finomított rendszerben a kérés küldése a megfelelő végponthoz függ a kérés típusától. Ha a típus egy STRING, akkor csak azokhoz a végpontokhoz lesz elküldve a kérés, amely kezelni tudja azt.

Algoritmus 14 Szabály 2 (finomított szabály a szemantikus böngészőkhöz)

```
 1: if fstate(f) = start_req && fworkingOn(f) = q then
 2:     for all e ∈ ENDPOINT do
 3:         if isIRI(q) && canIRI(e) || isString(q) && canString(e) then
 4:             if estate(e) = waiting then
 5:                 EXTEND REQUEST by req with
 6:                     reqQuery(req) := q
 7:                     eworkingOn(e) := req
 8:                     estate(e) := running
 9:                     rres(req) := undef
10:                 end extend
11:             end if
12:         end if
13:         fstate(f) = running
14:     end for
15: end if
```

4.4. SPARQL végpont választás névtér alapján [4]

4.4.1. Végpontok leírása

A federált rendszerek fontos része a végpont választási folyamat. A végpontokat általában a rajtuk elérhető adathalmaz valamely jellemzőjével írják le. A korábban az állítmányokat használták [45, 83], mert kis számosságban fordulnak elő az adathalmaz méretéhez viszonyítva, valamint a lekérdezésekben is ritkán szerepelnek változókként. Ezzel szemben a megoldásom az adathalmazban található névtereken alapszik. Egy adathalmazban szereplő névterek számossága szintén nem nagy méretű, és a legtöbb végponton a

[1] http://dbpedia.org/fct

névterek elérhetőek valamilyen formában. Viszont sok olyan prefix van, amely több végpont konfigurációjában is szerepelhet. Ilyen például az *rdf*, amely minden adathalmazban megtalálható. Ezek az állítmányok fognak nehézséget okozni a végpont választásban. A federált rendszer hármasmintákban szereplő információkból dönti el, mely végpont tudja kiértékelni őket. Az én megoldásomban a végpont választása a feltételben található prefixek és változók alapján történik. Ha ezek egyértelműek, vagyis minden hármasmintához egy végpontot választottunk, akkor kész vagyunk. Amennyiben nem egyértelmű, mert több végpont is tárol ilyen információt, akkor konfliktusba kerülünk, melynek feloldását mutatom most be.

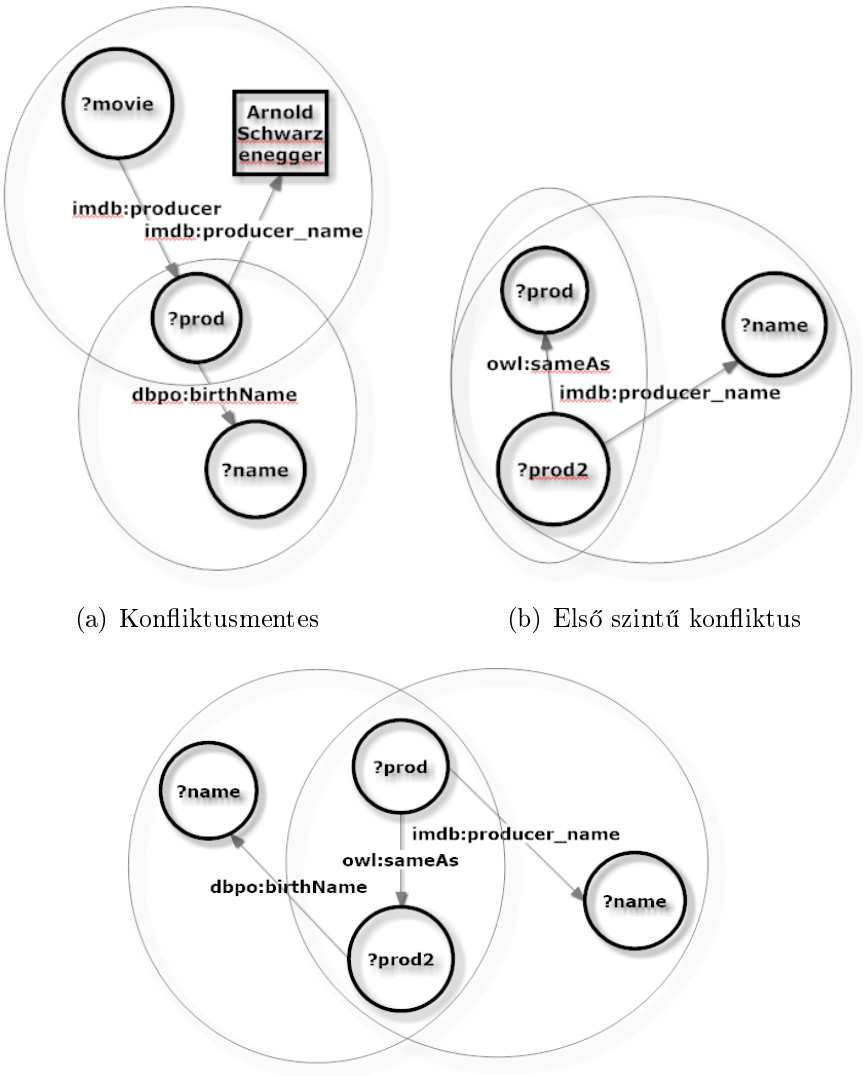
4.4.2. Konfliktus feloldás

A végpont választása egyértelmű, ha olyan prefixek szerepelnek, mint például a *<http://dbpedia.org/ontology>* (dbo) vagy a *<http://data.linkedmdb.org>* (imdb). Azonban lehetnek olyan feltételek is, amelyek több végponton is értelmezhetőek. Ilyenek például a *<http://www.w3.org/1999/02/22-rdf-syntax-ns#>* (rdf) vagy a *<http://www.w3.org/2002/07/owl#>* (owl). A kiértékelésnél ekkor több végpont is értelmezni tudja a feltételt. Emiatt ellenőrizzük a feltételben szereplő változókat, és megnézzük, hogy a többi feltételhez mely végpontok lettek kiválasztva. Ha a változók alapján találunk olyan hármasmintát, amit csak egy végpont képes kiértékelni, akkor ezt a hármasmintát is arra a végpontra fogjuk küldeni. Ha viszont azok a hármasminták is több végponton értékelhetőek ki, akkor másodszintű konfliktusról beszélünk, amelyre további feloldási megoldásra van szükség.

Ha a lekérdezés csak a következő sort tartalmazza (*?s rdf:type ?p*), akkor a rendszer nem tudja feloldani a konfliktusos helyzetet, mivel az *rdf:type* több végponton is értelmezhető és nincs más feltétel, amely alapján tudnánk szűkíteni a végpontokat, így nem tudjuk feloldani a konfliktust. Ebben segít Gorlitz [82] megoldása, amely minden végpontra kiküldené a lekérdezést, és az eredményeket uniózza.

A 4.1 ábrán konfliktusos helyzetekre láthatunk példákat. A 4.1(a) ábra olyan helyzetet mutat, ahol nincs konfliktus, mivel a lekérdezésben szereplő hármasminták egyértelműen kiértékelhetőek. A *?movie imdb:producer ?prod, ?prod imdb:producer_name 'Arnold Swarzenegger', ?prod dbpo:birtName ?name* hármasminták kiértékelhetőek a Linked Movie DataBase[2] és a DBpedia[3] végpontokon. A 4.1(b) ábrán első szintű hibákat láthatunk, ahol a *?prod2 owl:sameAs ?prod* hármasminta kiértékelése nem egyértelmű,

[2]http://data.linkedmdb.org/sparql
[3]http://dbpedia.org/sparql



(a) Konfliktusmentes (b) Első szintű konfliktus

(c) Másodszintű konfliktus

4.1. ábra. Konfliktusos hármasminták: egy lekérdezés akkor kerül konfliktusba, ha van olyan hármasmintája, amelyet nem tudunk egyértelműen eldönteni, hogy melyik végponton értékeljük ki. Az a) ábrán nincs konfliktus, hisz minden hármasminta egyértelmű. A b) ábrán az *owl:sameAs* konfliktusban van, de a *?prod2* változó olyan hármasmintához csatlakozik, amely nem konfliktus, így ezt a hármasmintát is ezen a végponton fogjuk kiértékelni. A c) ábrán a *owl:sameAs* van konfliktusban, de itt mind a két változója más végponton értékelődhet ki, így itt nem tudjuk feloldani a konfliktust.

hisz több végpont is ki tudja értékelni. Viszont a *?prod2 imdb:producer_ name ?name* hármasminta egyértelmű, hogy a Linked Movie Database-en értékelhető ki, így a konfliktusos hármasminta is ezen a végponton lesz kiértékelve. Az utolsó 4.1(c) képen egy másod szintű konfliktust láthatunk, ahol nem tudjuk eldönteni, hogy az adott hármasmintát hol értékeljük ki, emiatt minden végpontnak ki kell értékelnie azt.

A megoldásom első lépésként feldarabolja a lekérdezéseket hármasmintákra, majd a

4.1. táblázat. Kiválasztott végpontok

Sor	Hármasminta	Végpont(ok)
1	?actor2 owl:sameAs ?prod	**LinkedMDB**,**DBpedia**
2	?actor2 dbpo:birthDate ?birthDate	**DBpedia**
3	?prod imdb:producer_name "Arnold Schwarzenegger"	**LinkedMDB**
4	?movie imdb:producer ?prod	**LinkedMDB**
5	?movie dcterms:title ?movieTitle	DBpedia, **LinkedMDB**
6	?movie dcterms:date ?movieDate	DBpedia, **LinkedMDB**

névterek alapján meghatározza a megfelelő végpontokat. Először megtalálja azokat a hármasmintákat, amelyek konfliktus mentesek. Ezután keresi meg azokat, amelyek közös változón keresztül kapcsolódnak ezekhez a konfliktusmentes feltételekhez és első szintű konfliktusban vannak. Végül a másod szintű konfliktusos feltételeket a rendszer elküldi minden egyes végpontra és az eredményüket uniózza. Az 16. algoritmus a végpont meghatározás lépéseit írja le. A *tp* tárolja a hármasmintákat, amelyeket ellenőrizni szeretnénk, és hogy mely végpontok képesek megválaszolni azt a névterek alapján. Az e változó fogja tárolni a hármasmintákat az egyes végpontokhoz és v a változókhoz. A *conf_tp* tárolja azokat a hármasmintákat, amelyek konfliktusban vannak. Első körben megvizsgáljuk, hogy az egyes hármasmintához hány végpont található. Ha ennek a számossága egy, akkor az konfliktusmentes, és ki tudjuk értékelni az adott végponton. Ezt az információt letároljuk az e és v halmazokba. Ha több végpont is képes megválaszolni, akkor a *conf_tp* halmazba rakjuk, majd végig megyünk rajta és megpróbáljuk feloldani a konfliktust a korábbi információk alapján. Amennyiben ez sikerül, akkor az adott hármasmintát egy végponton tudjuk kiértékelni és tároljuk az e és v halmazban. Ha nem sikerül, akkor az összes végponthoz felvesszük. Az algoritmus végén az e fogja tárolni a hármasmintákat az egyes végpontokhoz. Ezeket fogja a rendszer lekérdezni és az eredményeket összegezni.

Példa 15 Federált példa lekérdezés

```
1: SELECT ?movieTitle ?movieDate ?birthDate
2: WHERE {
3:       ?actor2 owl:sameAs ?prod .
4:       ?actor2 dbpo:birthDate ?birthDate .
5:       ?prod imdb:producer_name "Arnold Schwarzenegger".
6:       ?movie imdb:producer ?prod ;
7:           dcterms:title ?movieTitle ;
8:           dcterms:date ?movieDate .
9: }
```

A 4.1 táblázatban a 15. példában szereplő lekérdezéshez kiválasztott végpontokat lát-

Algoritmus 16 Végpont meghatározás.

```
1: List<TriplePattern> tp = readServiceTriplePatterns();
2: Map<Endpoint,List<TriplePattern> > e = ;
3: Map<Variable,List<TriplePattern> > v = ;
4: List<TriplePattern> conf_tp = ();
5: foreach c in tp:
6:       if size(goodEndpoints(c)) = 1:
7:           push(egoodEndpoints(c), c)
8:           pushAll(vvariables(c), goodEndpoints(c))
9:       else:
10:          pushAll(conf_tp,c);
11:      end if
12: end foreach
13: foreach c in conf_tp:
14:      if canResolveConflit(c,e,v):
15:          push(eresolvedEndpoints(c), c)
16:          push(vvariables(c), resolvedEndpoints(c))
17:      else:
18:          pushAll(egoodEndpoints(c), c)
19:      end if
20: end foreach
```

hatjuk. Itt vannak olyan sorok, amelyek több végponthoz is tartozhatnak. A rendszer a változók alapján meghatározta, hogy mely végponton érdemes kiértékelni. Ezek szerepelnek félkövéren. Az első sor mindkét, a második sor a DBpedia végpontra, a 3-6 sorok pedig a LinkedMDB végpontra lesznek elküldve.

4.4.3. Kiértékelés

A végpont kiválasztási stratégiát nem teljesítményben, hanem használhatóságában ellenőriztem. A használhatósághoz két olyan speciális lekérdezést választottam, amely megfelelően szemlélteti a végpontkiválasztás használhatóságát. A lekérdezések kellően egyszerűek, ahhoz hogy a végpontok kezelni tudják.

4.2. táblázat. Lekérdezések futtatása a végpontokon

	DBpedia	LinkedMDB	FactForge (NYTimes)	Sparql.org	Saját rendszer
q1 Web Browser	Exception	0 row	-	Time out	-
q1 Code	Exception	0 row	-	Exception	2 row
q2 Web Browser	Exception	-	1 row	Time out	-
q2 Code	Exception	-	1 row	Exception	1 row

A használhatósághoz két lekérdezést használtam az egyik a FedBench [90] egy lekérdezése, ami a federált lekérdezésekhez készített benchmark (17. példa) (*q1*). Ez a

Példa 17 Fedbench Cross-Domain lekérdezés (q1)

```
1: SELECT ?pres ?party ?page
2: WHERE {
3:       ?pres rdf:type dbpedia-owl:President .
4:       ?pres dbpedia-owl:nationality dbpedia:United_States .
5:       ?pres dbpedia-owl:party ?party .
6:       ?x nytimes:topicPage ?page .
7:       ?x owl:sameAs ?pres .
8: }
```

lekérdezés a Cross-Domain lekérdezések közül az első, amely a DBpedia és NYTimes végpontokat kapcsolja össze. A másik a példában bemutatott lekérdezés (15. példa) (*q2*). Az ellenőrzéshez csak aktív végpontokat használtam (DBpedia, LinkedMDB, Sparql.org és FactForge [79]), amik mind elérhetőek voltak a mérés időpontjában (2014-01). Sok végponton vezettek be korlátokat, ami miatt a lekérdezések futtatása nem lehetséges. A lekérdezéseket a SPARQL végpontok webes felületén és programkódból a Jena [52] segítségével teszteltem. Az eredményeket a 4.2 táblázat tartalmazza. Az első sorban a Schwarzenegger lekérdezést futtattam a különböző webes felületeken. Ebben az esetben a kiértékelésnél alkalmaztam a SERVICE kulcsszót, ami megmondja, hogy a hármasokból melyiket hol kell kiértékelni. Értelemszerűen amikor a DBpedia-n futott a lekérdezés, akkor azok a sorok voltak a SERVICE részen belül, amelyek a LinkedMDB-n kellett hogy fussanak. A LinkedMDB-n való futtatás során pedig fordítva. A Sparql.org esetében két SERVICE részt használtam. A megoldásom esetében egyet sem. Az eredményekből láthatjuk, hogy a DBpedia kivételt dobott, amikor a lekérdezéseket akartuk futtatni akár a webes felületen, akár kódból. A kivételben szerepelt, hogy a lekérdezés túl költséges a DBpedia-nak. A LinkedMDB nem adott hibát, viszont nem is adta meg a helyes megoldást. Az eredményül kapott sorok száma 0 volt, ami félrevezető, hisz 2 sor volt az eredmény a *q1* lekérdezésnél. Olyan végpontok, amelyek nem tartalmazták az adatokat, mint a sparql.org, időkorlát miatt nem tudták megválaszolni a lekérdezést. Kódból való elérés esetén is jellemzően kivétellel állt le a futás. A FactForge volt az egyetlen végpont, amelynek félig sikerült megválaszolnia a lekérdezéseket. A *q1* esetében ő se tudott választ adni, de a *q2* esetében megkaptuk a megfelelő választ. A megoldásom mind a két lekérdezésben a megfelelő választ adta, ezzel mutatva meg, hogy a módszer működik.

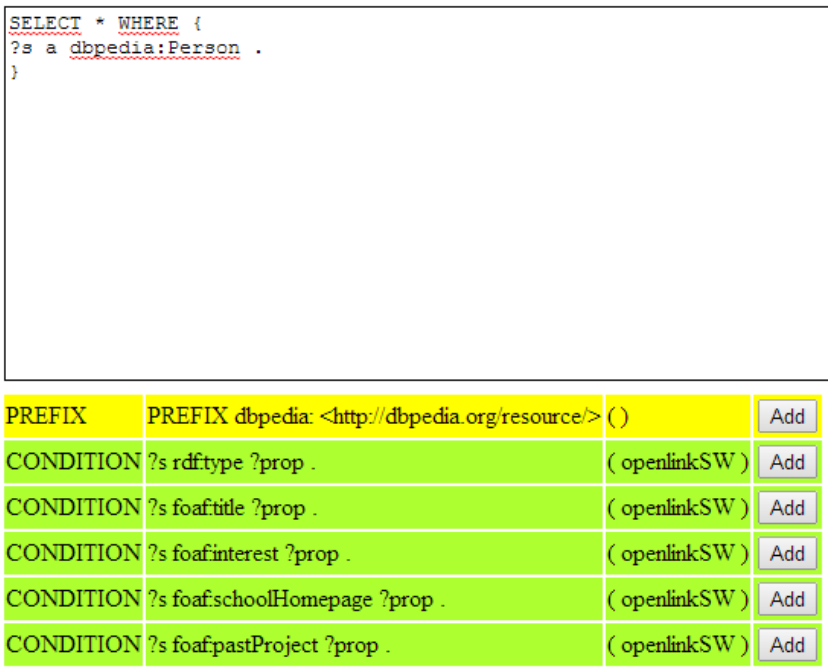
4.5. A SPARQL ajánlórendszer [5]

A másik probléma a SPARQL lekérdezések készítésével, hogy ismernünk kell az adathalmazokat, valamint az ott tárolt adatokat. A 4.3.1. fejezetben leírt modell erre a problémára ad egy megoldást, amely ajánlásokat tesz a felhasználónak a lekérdezés készítése közben. Erre a modellre készítettem egy prototípust, amely a fentiekben leírt funkciókat tudja megvalósítani. Az ajánlások készítése az előre beállított SPARQL végpontokon futtatott lekérdezésekkel történik, amik a következőek: factbook[4], dataGov[5], dblp[6], dbpedia[7], factforge[8], openlinkSW[9], linkedMDB[10], void[11]. Ezenkívül a rendszerbe be lett töltve az összes névtér rövid és hosszú formája. Ezeket a prefix.cc oldalról gyűjtöttük. A federált rendszer [108] azért előnyös, mert így a felhasználónak nem kell azzal foglalkoznia, hogy milyen végpontot válasszon a lekérdezése megírására, futtatására. A rendszer működése a következő. Ha változás van a lekérdezésben, akkor az egy háttérben futó alkalmazáshoz lesz elküldve aszinkron módon. Ez az értelmező az ARQ[12]-t használva a lekérdezésből kinyeri a WHERE-ben található feltételeket, és ajánlásokat készít a rendszer. Az ajánlások négy csoportba tartoznak: prefix ajánlás, típus ajánlás, állítmány ajánlás és a *rdfs:range* alapú állítmány ajánlás. Az ajánlás mind egy hármasminta, amely tartalmazza a lekérdezés valamely változóját.

A prefix ajánlás a lekérdezésben található névtereket ellenőrzi. Ha van a lekérdezésben olyan névtér, amely még nincs definiálva a lekérdezés elején, akkor a rendszer felajánlja, hogy vegyük hozzá a megfelelő prefixet. Ha olyan prefixet írtunk, amelyet a rendszer nem ismer, akkor egy hibaüzenetet kapunk, mely jelzi nekünk, hogy ezt a prefixet nekünk kell megadnunk. Ilyen lehet akkor, ha egy IRI-t más rövidítéssel használunk. A 4.2 ábrán láthatjuk a *dbpedia:Person* IRI-t amelyben a *dbpedia* a prefix. A rendszer ismeri ezt a prefixet és ajánlja, hogy vegyük hozzá a lekérdezéshez a 'PREFIX dbpedia: <http://dbpedia.org/resource>' sort.

A következő ajánlás a típus ajánlás. Egy SPARQL lekérdezés általában tartalmaz olyan információt, amely egy adott változó típusára (*rdf:type*) jelent megszorítást. Egy lekérdezés megírásának első lépéseként szoktuk ezt kiválasztani. Emiatt a rendszer lekérdezi a lehetséges típusokat a beállított végpontokról és azokat ajánlja a felhasználónak.

[4] http://wifo5-04.informatik.uni-mannheim.de/factbook/sparql
[5] http://services.data.gov/sparql
[6] http://dblp.rkbexplorer.com/sparql
[7] http://dbpedia.org/sparql
[8] http://factforge.net/sparql
[9] http://lod.openlinksw.com/sparql
[10] http://data.linkedmdb.org/sparql
[11] http://void.rkbexplorer.com/sparql
[12] http://jena.sourceforge.net/ARQ/



4.2. ábra. Sparkql ajánlórendszer pilot felülete ajánlással. A rendszer a hármasminta alapján prefixet (*dbpedia*), és állítmányokat ajánl a *?s* változóra a típus (*dbpedia:Person*) alapján.

Ha a felhasználó írt vagy kiválasztott valamilyen hármasmintát, akkor a rendszer képes neki állítmányokat ajánlani az adott változóra. Ehhez az kell, hogy ismerjük az adott változó típusát, ami alapján le tudjuk kérdezni, hogy ezek az entitások milyen állítmányokkal rendelkeznek.
Az utolsó ajánlás az *rdfs:range* alapú állítmány ajánlás. Az *rdfs:range* az állítmányról mondja meg, hogy milyen típusúnak kell lennie a hármasmintában a tárgynak. Ezzel az információval újabb ajánlásokat tudunk tenni, hisz ha a felhasználó megírt, kiválasztott valamilyen hármasmintát és annak a tárgy helyén változó van, akkor a végpontokról le tudjuk kérdezni a típusát. Ezután a típus ajánlással további hármasmintákat tudunk készíteni.

Az ajánlásokat egyszerű SPARQL lekérdezésekkel tudjuk kinyerni a végpontokból. Ezeket láthatjuk a 4.3. táblázatban. A típusok lekérdezéséhez a *TypeQuery*-t alkalmaztam, amely egyszerűen lekérdezi a lehetséges típusokat egy adott végpontról. Ezután következnek az állítmány ajánlásokhoz szükséges lekérdezések. A *PredicateAndIdentityQuery* és a *PredicateAndRangeQuery* lekérdezések egy adott típushoz kérdeznek le maximum 500 különböző állítmányt. A lekérdezések egyetlen paramétere egy típus, illetve egy állítmány. A lekérdezés csak azokat az állítmányokat kérdezi le, amelyek nem az

TypeQuery :
```
SELECT DISTINCT ?type WHERE {
    _ rdf:type ?type .
}
```

IdentityQuery :
```
SELECT DISTINCT ?id WHERE {
  ?id rdf:type <sometype> . } LIMIT 10
```

PredicateQuery :
```
SELECT DISTINCT ?p WHERE {
  <id> ?p _ . }
```

IdentityRangeQuery :
```
SELECT DISTINCT ?id WHERE {
  <someproperty> rdfs:range ?range .
  ?id rdf:type ?range . } LIMIT 10
```

PredicateAndIdentityQuery :
```
SELECT DISTINCT ?p WHERE {
  ?s a <some:type> .
  ?s ?p [] .
  FILTER(?p != rdf:type)
} LIMIT 500
```

PredicateAndRangeQuery :
```
SELECT DISTINCT ?p WHERE {
  <some:property> rdfs:range ?range .
  ?s a ?range .
  ?s ?p [] .
  FILTER(?p != rdf:type)
} LIMIT 500
```

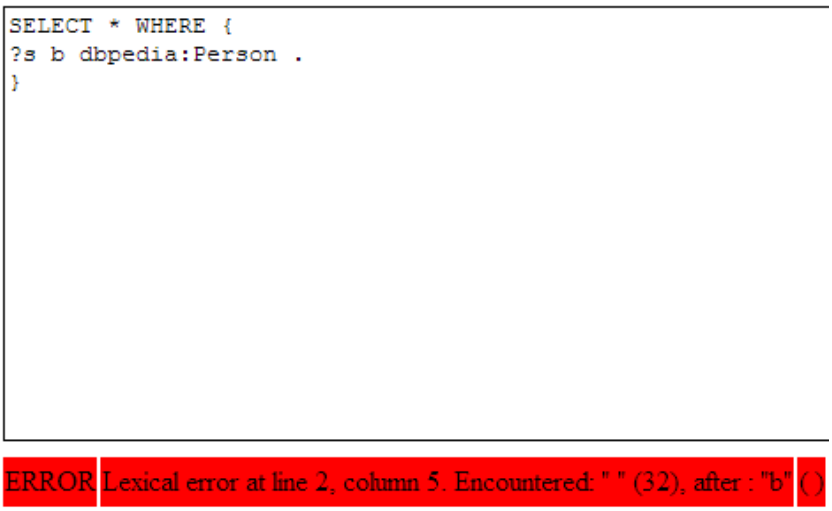
4.3. táblázat. SPARQL lekérdezések az ajánlásokhoz

rdf:type. A sok objektum lekérdezése viszont sok végpontnak nem megoldható és nem kapunk eredményt, emiatt ezt nem alkalmaztam, hanem helyette két egyszerűbb lekérdezéssel kérdeztem le az állítmányokat. Az első az *IdentityQuery*, amely lekérdez 10 entitást az adott típusra. A 10 kellően elég ahhoz, hogy a rendszer gyorsan tudjon működni és elég ajánlást biztosítson. A második a *PredicateQuery*, amely az adott entitásokról kérdezi le az állítmányaikat, melyeket a rendszer összegyűjt, majd kiszűri a duplikátumokat és elkészíti az ajánlásokat. Az *rdfs:range* alapú ajánlásnak az első lépése, hogy lekérdezzük az állítmány típusát és a típus alapján ajánlunk állítmányokat. Az entitások lekérdezését az *IdentityRangeQuery* végzi, amelynek egy állítmányt kell megadni és visszaad nekünk 10 entitást. Ezután ismét alkalmazhatjuk a *PredicateQuery* lekérdezést.

A típus ajánlás folyamatát láthatjuk a 18. algoritmusban. Az algoritmus végig megy minden beállított végponton és kinyeri a lehetséges típusokat, és egy ajánlás listába pakolja az így elkészült hármasokat. A 19. algoritmus az állítmányok ajánlásának folyamatát mutatja be. Hasonlóan az előbbi algoritmushoz végigmegyünk az összes végponton, majd minden végpontról lekérdezünk 10 entitást, majd lekérdezzük az állítmányaikat. Az algo-

Algoritmus 18 Típus ajánlás

```
1: function RecommendTypes( )
2:     RecommendList = List < triple >
3:     for all e ∈ Endpoints do
4:         for all type ∈ runTypeQuery(e) do
5:             push(RecommendList,
                       (?s, rdf : type, type))
6:             storeEndpointInfo(getInfo(type))
7:         end for
8:     end for
9:     return RecommendList
10: end function
```



4.3. ábra. Sparkql ajánlórendszer pilot felülete hibajelzéssel. A lekérdezésben szereplő 'b' karakter se nem változó, se nem egy prefix, így a rendszer hibának jelzi.

ritmus paramétere egy változó és egy típus. A változó az elkészült hármasmintákban fog szerepelni, így kapcsolva össze a korábbi hármasmintákkal. Az összegyűjtött és redukált állítmányokból végül elkészítjük az ajánlásokat. Az ajánlás alanya a változó, az állítmány a kinyert tulajdonság és a tárgy egy újabb változó, amelyet a kapott változóból és az állítmányból készítünk el.

A megoldásomról egy pilot rendszer is készült, melynek felületét a 4.3 és 4.2 ábrák mutatják. Az első ábrán azt látjuk, hogy az alkalmazás ellenőrzi, hogy az adott SPARQL lekérdezés helyes-e. Az ábrán szereplő lekérdezés állítmánya egy 'b', amely érvénytelenné teszi a lekérdezést. A második ábra viszont már az ajánlásokat jeleníti meg. A lekérdezés hármasmintája az '?s' változóra tett feltétel amely a típusát mondja meg. Az '?s' változónak *dbpedia:Person*-nak kell lennie. Ebből az egy sorból a rendszerem névtér ajánlást tesz a *dbpedia*-ra és állítmány ajánlásokat az '?s' változóra.

Algoritmus 19 Állítmányok ajánlása

```
1: function RecommendPredicates(?x, type)
2:     RecommendPredicate = Map < Predicate >
3:     for all e ∈ Endpoints do
4:         for all entity ∈ getEntityInThisType(type) do
5:             for all prop ∈ getPropFromEntity(entity) do
6:                 push(RecommendPredicate, prop)
7:             end for
8:         end for
9:     end for
10:    for all prop ∈ RecommendPredicate do
11:        push(RecommendList, (?x, prop, ?x_prop))
12:        storeEndpointInfo(getnInfo(prop))
13:    end for
14:    return RecommendList
15: end function
```

4.6. Ajánlások használata federált rendszereknél [8]

4.6.1. Költség modell a federált lekérdezésekhez

Ebben a fejezetben bemutatok egy költség modellt a federált lekérdezések kiértékelésére. A modell a kiértékeléséhez szükséges lekérdezések számát adja meg. Elsőnek bemutatom az általános modellt, majd hogy hogyan értelmezhető ez az ASK és a katalógus technikánál. A modell tartalmaz egy beállítási és egy végpont választási részt. A beállítási rész azokat a lekérdezéseket tartalmazza amelyek szükségesek ahhoz, hogy a rendszer működéséhez szükséges információkat megszerezzük. A végpont választási rész pedig azokat a lekérdezéseket tartalmazza, amelyek a kiértékelés során szükségesek.

$$Cost_{MaxEval}(n) = Cost_{conf}(n) + Cost_{subQuery}(n) \tag{4.1}$$

$$Cost_{conf}(n) = (\sum_{i=1}^{n} Cost_{subConf}) * count(triple) \tag{4.2}$$

$$Cost_{subQuery}(n) = (\sum_{i=1}^{n} Cost_{EPQuery}) * count(subQuery) \tag{4.3}$$

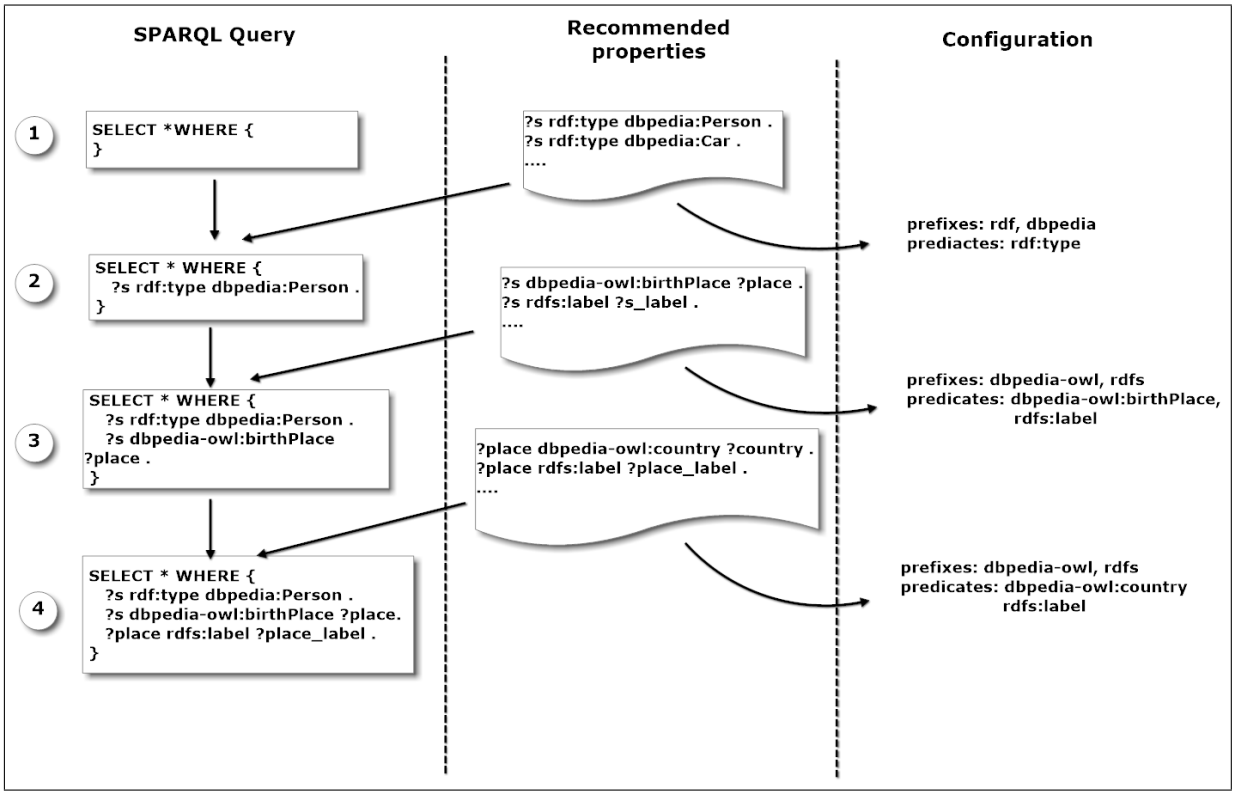
Legyen a végpontok száma n. Ekkor a 4.1. egyenlet a maximális kérések számát mutatja, ami egy kiértékeléshez szükséges. A költség függ attól, hogy hány végpontot (n) használunk. A legrosszabb esetben minden végpontot be kell állítanunk és minden végpontot le kell kérdeznünk a kiértékelés közben. A képlet első tagja (4.2. egyenlet) tar-

talmazza annak a költségét, hogy hány lekérdezés szükséges a végpontok beállításához. A rendszer valamilyen információt gyűjt a végpontokról ($Cost_{subConf}$), amelyeket a végpont választásnál fogunk használni. A képlet második tagja (4.3. egyenlet) az al-lekérdezések ($Cost_{EPQuery}$) kiértékelésének költségét mutatja be. A legrosszabb esetben minden végpontra szükségünk van, és minden al-lekérdezés fut minden végponton. Ez persze nem jellemző hisz a végpont választás célja, hogy kiválasszuk azokat, amelyek szükségesek a kiértékeléshez. A kutatásom célja, hogy csökkentsem a lehetséges végpontoknak a számát, aminek az alapja az előző fejezetben bemutatott ajánlórendszer. Az ötlet az, hogy minden ajánlással egy információt kapunk az adott végpontról, ami elég információ ahhoz, hogy ki tudjuk értékelni a lekérdezést. Minden iterációban, amikor a rendszer új ajánlásokat kap, ki tudja nyerni az abban szereplő állítmányokat, vagy épp a prefixeket. Ezek az információk mindig a legfrissebbek, hisz a működő végpontokról érkeznek. Másik előny, hogy ha valamely végpont nem válaszol, akkor azokat a végpontokat ki tudjuk hagyni a kiértékelésből. Ezen információk segítségével csökkenteni tudjuk a lehetséges végpontok számát. A kiértékeléshez meglévő federált rendszereket fogok alkalmazni úgy, hogy a konfigurációjukat megváltoztatjuk az ajánló rendszerből nyert információk alapján.

4.6.2. Ajánlórendszerből kinyert információ

Az előző fejezetben bemutatott algoritmusok (18. algoritmus , 19. algoritmus) új hármasmintákat készítenek a félig megírt lekérdezésekhez. Amikor egy végpont válaszol az ajánlórendszernek, akkor információt is megoszt az ott tárolt adatokról. Amikor az állítmány ajánlás készül, akkor megkapjuk, hogy ezek a végpontok az adott állítmányt ismerik és ki tudják értékelni. Ezenkívül a válaszok tartalmaznak névtereket, amelyek szintén jól leírják az adott végpontot. Korábban említettem, hogy ezek az információk szükségesek a federált rendszereknek, hogy ki tudják választani a megfelelő végpontokat a kiértékeléshez. Emiatt ezeket az információkat eltároljuk és felhasználjuk a kiértékelés során.

A 4.4. ábrán láthatunk egy példát az ajánlások készítésére. A bal oldalon láthatjuk a SPARQL lekérdezés fejlődését, amely az első lépésben még nem tartalmaz hármasmintát. A rendszer ekkor a végpontokról kapott típusokat ajánl a felhasználónak (az ábrán: *dbpedia:Person, dbpedia:Car*), amelyből állítmány (*rdf:type*) és névtér (*rdf, dbpedia*) információkat tudott kinyerni (ábra jobb oldala). A következő lépésben a felhasználó kiválasztja a *?s rdf:type dbpedia:Person* hármasmintát. Ekkor újabb ajánlásokat fog kapni, mivel tudjuk az *?s* változó típusát. Ehhez a hármasmintához tartozó ajánlásokból (*dbpedia-owl:birthPlace, rdfs:label*) a rendszer ismételten letárolja az állítmány és névtér



4.4. ábra. SPARQL készítés ajánlással és információ kinyeréssel. A képen az ajánlás folyamatát látjuk. Baloldalon látható a lekérdezés készítésének lépései. Középen láthatjuk azokat a hármasmintákat, amelyeket a rendszer ajánlott a felhasználónak, jobb oldalt pedig az ajánlásokból kinyert prefixek és állítmányok láthatóak, amelyeket a végpontkiválasztásnál fogunk felhasználni.

információkat. Ezután ha kiválasztjuk a *?s dbpedia:birthPlace ?place* hármasmintát, akkor az *rdfs:range* állítmány segítségével kérdezünk le információkat a *?place* változóra. Megkapjuk, hogy a *?place* változónak a típusa *dbpedia:Place* kell hogy legyen. Ennek a típusnak is lekérdezzük az állítmányait, amit letárolunk és hármasmintákat készítünk belőlük.

1. Lemma (Az ajánlórendszerből nyert információ elég a kiértékeléshez). *Legyen EPQ azoknak a végpontoknak a halmaza, amelyek szükségesek a kiértékeléshez. Legyen Endpoints az összes olyan végpontnak a halmaza, amit ismer a rendszer. Legyen a UserSelect egy lista, ami a felhasználó által választott hármasmintákat tartalmazza. Ekkor* $\exists$ *tp* $\in$ *UserSelect, e* $\in$ *Endpoints, answer(e,tp)* $\Rightarrow$ *e* $\in$ *EPQ*

1. Bizonyítás. *A federált rendszernek minden hármasmintához (TP) szüksége van legalább egy végpontra. Ha a TP-t tartalmazza a végpont, akkor a végpont meg tudja válaszolni azt. Bizonyítsuk ezt indirekt módon. Legyen e egy végpont, ami meg tudja válaszolni TP-t, de e ne szerepeljen az EPQ halmazban. Ez azt jelenti, hogy TP nem erről a végpontról lett*

készítve, de a felhasználó kiválasztotta. Minden TP egy végpontról kell hogy jöjjön, tehát TP-nek is e-ről kell hogy jöjjön, és e-nek meg kell tudja válaszolnia TP-t és szerepelnie kell az EPQ-ban.

4.6.3. Költség modell a federált rendszerekhez

A 4.6.1 fejezetben bemutattam egy általános modellt a federált SPARQL kiértékeléshez, most pedig ismertetem a költségeit a két végpont választási technikának (ASK, katalógus).

Költség modell az ASK lekérdezéses technikához

Az ASK lekérdezésekkel történő végpont választás minden alkalommal lekérdezi az összes végpontot, ami egyenlő a korábban bemutatott konfigurációs költséggel ($Cost_{conf}$). A kiértékeléshez viszont már nem fogja az összes végpontot használni, mivel nem fog minden végpont igaz állítással visszatérni a konfiguráció során. Emiatt ez a költség ($Cost_{subquery}$) kevesebb lesz.

Költség modell a katalógus technikához

A katalógus technika előre beállított információkkal rendelkezik a végpontokról. Ez a fázis független a kiértékeléstől, emiatt a konfigurációs költség 0. Ebben az esetben a rendszer csak azokat a végpontokat használja, ami a konfigurációban szerepel és megfelel a lekérdezéshez. Emiatt a kiértékelési rész költsége a kiválasztott végpontoktól függ, ugyanúgy mint az ASK módszernél.

Költség modell az ajánló rendszeres kiértékeléshez

Az ajánlórendszer egyszerű lekérdezéseket használ a lekérdezés elkészítéséhez, emiatt mindig friss információval rendelkezik. Az előnye ennek, hogy ezzel csökkenteni tudja a lehetséges végpontoknak a számát, mivel csak azok a végpontok kellenek, amelyektől jött a hármasminta. Itt a konfigurációs részt a lekérdezés készítő ajánlórendszer végzi, amely minden végpontot lekérdez minden állítás készítésekor. A kiértékelés viszont a korábbi technikákkal történik.

Költség összehasonlítások

Legyen $recommend(n) \leq n$, ahol a $recommend(n)$ az ajánló rendszer által használt végpontoknak a száma.

$$Cost_{catalog}(recommend(n)) \leq Cost_{catalog}(n) \tag{4.4}$$

$$Cost_{ASK}(recommend(n)) \leq Cost_{ASK}(n) \tag{4.5}$$

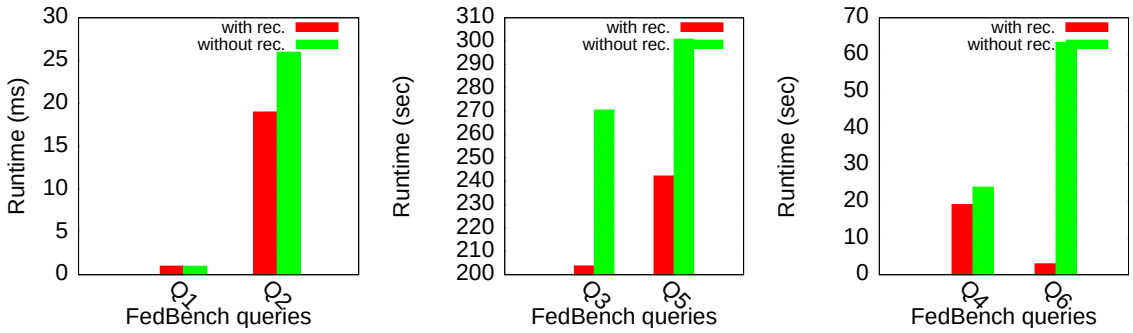
Amikor a katalógus technikát alkalmazzuk a rendszer végig olvassa a konfigurációs fájlt, és csak azokat a végpontokat kérdezi le, amelyek meg tudják válaszolni a hármasmintát. Ahogy korábban említettem, nem változtattam meg a federált rendszer működését, csak a konfigurációs fájlt, ahol csak azokat a végpontokat hagytam benne, amelyek az ajánlások során használatban voltak. A 4.4. képlet azt mutatja, hogy a költség kevesebb, de legrosszabb esetben ugyanannyi, mintha nem használnánk az ajánlórendszert. A legrosszabb esetben az összes végpont konfigurációjára szükségünk van.

Amikor az ASK technikát használja a rendszer, akkor minden végpontot ellenőriz, hogy képes-e megválaszolni a hármasmintát. A költsége az ASK lekérdezéseknek több, mint a katalógus technikának, mivel itt a konfigurációs fázisban is lekérdezzük a végpontokat. A 4.5. képlet mutatja, hogy ajánlórendszerrel a költség kevesebb vagy ugyanannyi, mint nélküle. Amikor az ajánlórendszer által kinyert információkat alkalmazzuk az ASK technikánál, akkor kevesebb végpontot kell a rendszernek lekérdeznie, de legrosszabb esetben az összeset.

4.6.4. Mérési eredmények

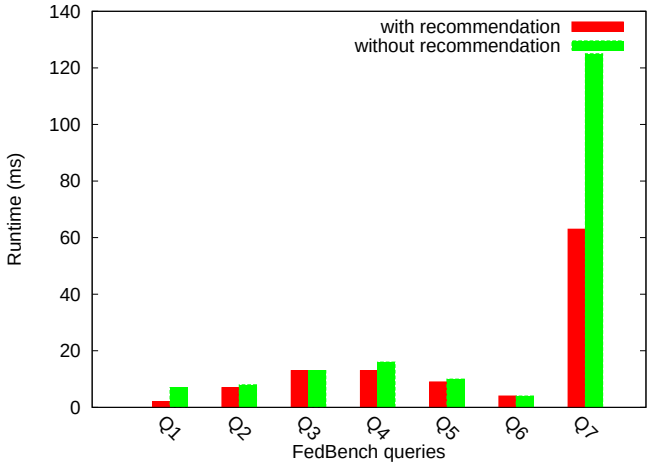
Az modellt két korábbi federált rendszerrel vizsgáltam meg. Az első rendszer, amit használtam a Darq [45], amely a katalógus technikát alkalmazza. A rendszer használata előtt egy konfigurációs futás szükséges, amely elkészít egy katalógus fájlt. A másik rendszer a FedX [91], ami az ASK technikát alkalmazza a federált lekérdezések kiértékelésére. A méréshez használt gép Intel Xeon X5650 CPU (4 core, 2,67 GHz) processzorral és 4GB memóriával rendelkezett. Mivel a SPARQL végpontok elérhetősége befolyásolhatta volna a mérést, emiatt két Intel Core i5 650 CPU (4 core, 3,2GHz) processzorral és 4GB memóriával rendelkező gépre telepítettem Virtuoso [67] (ver. 06.01) végpontokat. Mind a két végpont a FedBench [90] adathalmazokat tárolta. Az első végpont tárolja a NyTimes és a Jamendo adathalmazokat, a második pedig a DBpedia, LinkedMDB és a Geonames adathalmazokat. A Virtuoso-kat úgy állítottam be, hogy az adathalmazokat el lehessen érni külön-külön is. Ehhez az kellett, hogy az adathalmazok külön gráfként legyenek tárolva. A mérésekhez a FedBench [90] Cross-Domain lekérdezéseket használtam. Mindegyiket ötször mértem meg egy warm-up futás után.

A 4.5. ábra mutatja a mérések eredményét a Darq rendszeren az ajánlórendszerrel és anélkül. A bal oldali ábra mutatja az Q1 és Q2 lekérdezés eredményét, a középső a Q3 és Q5 lekérdezés eredményét, a jobb oldali pedig a Q4 és Q6 lekérdezések eredményét. A Q7 lekérdezés mind a két esetben memória hibával nem adott eredményt. Láthatjuk, hogy minden esetben az ajánlórendszer által gyűjtött információval hatékonyabban tudtuk kiértékelni a lekérdezést.



4.5. ábra. DarQ federált lekérdezés mérési eredménye a LUBM lekérdezésekre. Az ábrán az ajánlórendszerből kinyert információk alapján szűkített végpontokon történő kiértékelés és az összes végpontot használó kiértékelés látható.

A 4.6. ábrán a FedX rendszerrel végzett mérések láthatóak. Az ábrán látható, hogy minden lekérdezés gyorsabban fut, ha használjuk az ajánlórendszer által kapott információkat kivéve a 3 és 6 lekérdezéseknél. Itt az eredmények ugyanazok, hisz a lekérdezés minden végpontot használ a kiértékeléshez.



4.6. ábra. FedX federált lekérdezés mérési eredménye a LUBM lekérdezésekre. Az ábrán az ajánlórendszerből kinyert információk alapján szűkített végpontokon történő kiértékelés és az összes végpontot használó kiértékelés látható.

4.7. Öszefoglalás és kapcsolat a tézisekkel

Ebben a fejezetben a szemantikus federált rendszerekkel elért eredményeimet mutattam be. A federált kiértékelés olyan módszer, ahol nem mondjuk meg, hogy az adott hármasmintát mely végponton szeretnénk kiértékeltetni, hanem a rendszer dönti el azt. A döntéshez vagy mindig megkérdezi a végpontot (ASK technika), vagy valamilyen információt tárol róla (katalógus technika), ami alapján tud dönteni. Az első részben az ASM modell segítségével írtam le a federált rendszerek működését, melyet finomítottam, két konkrét esetre. Az egyik eset a szemantikus böngészők szerver oldali leírása, még a másik eset a SPARQL ajánló rendszer. Ez segít a SPARQL-ben nem jártas felhasználóknak szemantikus lekérdezéseket írni. Ezek a 2. tézisemhez kapcsolódnak, amely a SPARQL ajánló rendszerről és annak az ASM segítségével történő formális leírásáról szól. A fejezetben bemutattam hogy az ajánlásokkal együtt információkat is kapunk a végpontokról, amelyeket fel tudunk használni a federált kiértékelés során. Ezek az információk lehetnek akár a névterek is. A fejezet ezen része a 3. tézisemhez kapcsolódik.

5. fejezet

Szemantikus adatok és az osztott rendszerek kapcsolata

A következő fejezetben az osztott rendszerekre és a Big Data eszközökre térek át. A szemantikus adatok tárolása egy fontos kérdés ebben a témakörben. Ahogy említettem a szemantikus adatbázisok nagy méretűek és ezenkívül strukturálatlan szerkezetűek, ami miatt a tárolásuk és lekérdezésük nem egyszerű. Ezek a tulajdonságok (méret, strukturálatlanság) a Big Data témakörben található adatok fő jellemzői. Emiatt jelentek meg olyan eszközök, amelyek a nagy és nehezen kezelhető adatok elemzésére szolgálnak. Kutatási kérdésként merül fel, hogy hogyan lehet használni ezeket az eszközöket a szemantikus adatoknál. A 5.3 alfejezetben a Hadoop rendszert fogjuk alkalmazni arra, hogy csökkenteni tudjuk az adatok méretét olyan szintre, amelyet már egy egyszerű gráfvizualizációs eszköz meg tud jeleníteni. A megjelenítés segítségével hasznos információkat kaphatunk az adathalmazról, így könnyebben tudunk megfelelő lekérdezést írni. Az itt bemutatott eredményen Rácz Gáborral dolgoztam együtt, és az én részem a biszimuláció Big Data rendszerekre való tervezése és hatékony megvalósítása volt. A 5.4 alfejezetben pedig a Spark GraphX nevezetű gráf elemző rendszerén mutatom meg, hogyan lehet kiértékelni egy szemantikus lekérdezést. Végül bemutatom az algoritmus egy javított változatát. Ezek az eredmények főként önálló érdemek, melyben segítségemre volt Rácz Gábor.

5.1. Bevezetés

A szemantikus adatoknak az összekapcsolásában nagy lehetőség rejtőzik, hisz találhatunk az információk között olyan kapcsolatokat, amelyeket eddig nem ismertünk. Vegyük például egy biológiai és egy kémiai adathalmazt, amelyek összekapcsolásával egy biológiai

anyag kémiai összetevőiről is kaphatunk információt [74].

Másfelől az adatok változatossága és mérete viszont kihívást jelent a kutatók számára is. Egy szemantikus adatban lévő kapcsolatok feltárásához nem csak következtető rendszerek alkalmazhatóak, hanem vizualizációs eszközök is. Egy nagy gráf megjelenítése viszont átláthatatlan és értelmezhetetlen. Ebben a fejezetben bemutatunk egy olyan megoldást, ahol a MapReduce [58] paradigmát alkalmazzuk a szemantikus adatokra. A célunk, hogy meg tudjuk jeleníteni az adatok struktúráját szűrésekkel és összevonásokkal.

Az itt bemutatott újszerű módszer csökkenti a szemantikus adatok méretét és komplexitását. Az algoritmusnak két bemenő paraméterre van szüksége, hogy mely állítmányok azok, amelyeket össze lehet vonni és csak a számosságuk fontos, vagy melyek azok, amelyeket meg szeretnénk jelenteni. A megjelenítésre szánt állítmányok az adatok struktúráját fogják megmutatni. Ezeket a paramétereket nevezzük *lánc* tulajdonságnak, mert az adatokban ezek láncként szerepelhetnek egymás után. Gondoljunk itt olyan tulajdonságra például, mint a 'szülő'. Egy 'szülő'-nek is van szülője, amivel egy családfát tudunk megjeleníteni. A másik tulajdonság a *nem-lánc* tulajdonság, ahol azt jelenítjük meg, hogy az egyes elemből mennyi szerepel ilyen állítmánnyal. Ha veszünk egy olyan állítmányt például, hogy 'autó tulajdonosa', és nem vagyunk kíváncsiak magukra az autókra, akkor ezeket összevonva jelenítjük meg. A kettőt tulajdonságot együtt alkalmazva egy olyan családfát láthatunk, ahol minden csomóponthoz van egy él, hogy hány autója van. A fő eredményünk az összevonási módszer, amely az éleket redukálja a gráfban. A kimenetet GraphML [73] formátumba készítjük el, amely egy szabványos XML alapú formátum gráfok tárolására és a legtöbb gráf megjelenítő alkalmazás képes kezelni. Az így kapott eredményt végül a yEd[1] eszköz segítségével jelenítettük meg.

Az algoritmusunkat az Apache Hadoop [98] rendszeren alkalmaztuk, amely egy open-source megvalósítása a MapReduce [58] technikának. Az eredmények kiértékeléséhez a DBpedia [46] és a Freebase [57] adathalmazokat használtuk különböző *lánc* és *nem-lánc* tulajdonsággal. A DBpedia a Wikipedia-ból kinyert adatokat tárolja szemantikus formában, még a Freebase egy olyan tudásbázis, amelyet a saját közösségük kezel. Az eredmények rávilágítottak olyan problémákra is, amelyek az adathalmaz minőségét jellemzik. Ilyenek például, hogy egyes élek hiányoznak, vagy rossz csomóponthoz kapcsolódnak, vagy épp rossz a címkéjük.

Egy másik kihívás a szemantikus adatok lekérdezése. Az adatok komplexitásából adódóan, ha egy bonyolultabb lekérdezést szeretnénk megválaszolni, akkor a mai rendszerek nehezen boldogulnak vele. Egy relációs táblában gráf jellegű adatokat tárolni és lekérdez-

[1] http://www.yworks.com/en/products_yed_about.htm

ni nem hatékony, emiatt több kutató is azzal kezdett el foglalkozni, hogy hogyan lehet a szemantikus lekérdezéseket megválaszolni Big Data eszközökkel. Az első ilyen eszköz a Hadoop volt, melyet több olyan eszköz is követett, amely a Hadoop-ot alkalmazza háttérnek, de más feladatokat is el tud látni. Ilyen eszköz például az Pig vagy a Hive, amelyek főként lekérdezőnyelvet biztosítanak az adatokhoz. A legnagyobb problémája ennek a rendszernek viszont, hogy az átmeneti eredményeket a háttértáron tárolja, emiatt nagy az I/O műveletigény egy alkalmazás futtatásához. Ez nagyban lassítja az adatok feldolgozását, amely egy lekérdezésnél probléma. A Berkley egyetemen viszont kifejlesztettek egy olyan keretrendszert, amely megpróbálja a memóriában tartani az adatokat mindaddig, amíg szükség van rájuk. Ez a rendszer a Spark [78], amelynek a fő komponense a memóriában elosztott objektumok: a Resilient Distributed Dataset (RDD) [99]. További előnye az eszköznek, hogy olyan területeken is lehet használni a kiegészítőjük segítségével, amelyekre a Hadoop nem volt képes. Ilyen területek a stream alapú feldolgozás, a gépi tanulás, vagy ami számunkra fontos a gráf feldolgozás. A gráf feldolgozó eszközt nevezik GraphX [114] -nek. Az osztott gráf feldolgozás problémája, hogy hogyan tároljuk a gráfot egy klaszterben, és a feldolgozásnál hogyan tudunk hozzáférni ezekhez az adatokhoz. A másik probléma, hogy a gráf feldolgozás iteratív folyamat, ami a párhuzamosítást nehezíti meg. A megvalósítás alapja a Bulk Synchronous Parallel (BSP) [19] modell, amely a számítások párhuzamosítására adott egy megoldást processzorok segítségével. A BSP három lépésből tevődik össze. Az első lépésben a processzorok számítást végeznek, majd a második lépésben elküldik az eredményüket a többi processzornak, végül a szinkronizációs lépés következik, hogy minden processzor befejezze a futását és minden üzenet megérkezzen a megfelelő helyre. Ezután indul a következő iteráció, ahol a processzorok a kapott üzenetek alapján újra számítást végeznek és az eredményt elküldik a többi processzornak, és így tovább. Ezen modellen alapszik a Pregel [76] modell, ahol a processzorok a gráfok csúcsai. Minden csúcsnak van egy saját értéke, amely változik a számítás során, és ezt az értéket küldi el azoknak a csomópontoknak, amelyek az ő szomszédai. Erre a modellre készítettem el a Sparkql[2] -t [9], amely meg tudja válaszolni a SPARQL lekérdezéseket, majd javítottam a működést a P-Sparkql[3]-ben [10].

5.2. Korábbi munkák

A VoID [64] egy RDF Schema szóhalmaz, amelynek a célja hasonlít a redukáló rendszerünkhöz csak más megközelítésből. Az adathalmaz megismeréséhez leírásokat használ,

[2]https://github.com/ggombos/Sparkql
[3]https://github.com/ggombos/PSparkql

amelyek a következő típusokba sorolhatóak:

- általános metainformációk, mint név vagy leírás,
- metódusok, amivel az adathoz hozzá tudunk férni (SPARQL végpont, lementett adat, ...),
- strukturális leírások, ami egy magasabb szintű rálátást ad az adathalmazra,
- összekapcsolhatósági lehetőségek más adathalmazokkal, amit ebben az adathalmazban használnak.

A strukturális leírás statisztikai adatokat ad meg az adathalmazról, mint például az osztályok, állítmányok számosságát. Ez az adat jó kiindulási adat a mi redukáló rendszerünknek is, hisz a gyakori állítmányok jól alkalmazhatóak a *lánc* tulajdonságnál, ugyanis ezek jól jellemzik az adathalmazt.

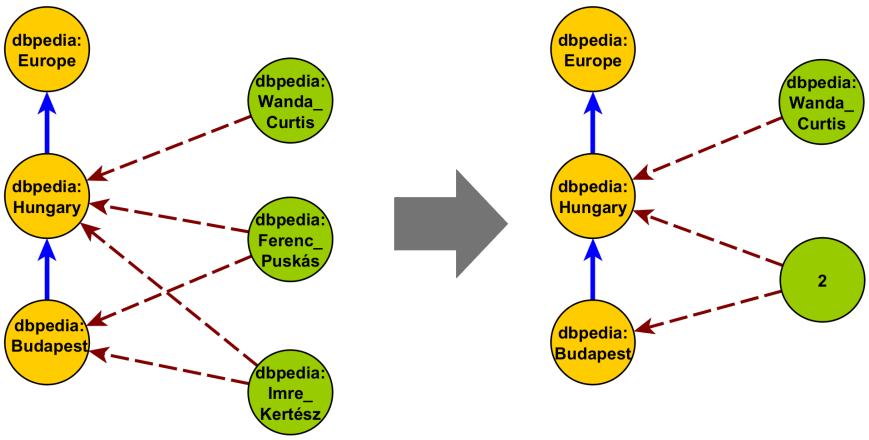
A rendszerünk az előre definiált tulajdonságokon alapszik. A kimenete egy olyan gráf, amit egy gráfmegjelenítő eszközzel kezelünk, hisz a vizualizációs eszközök segítik az adathalmaz megértését. Több ilyen vizualizációs eszköz is létezik: PGV Explorer [49], LODmilla [116], vagy a VizBoard [97]. A PGV Explorerrel a gráfot bejárva láthatjuk az adatokat, ami annyit jelent, hogy kezdetben egy kis részét látjuk csak az eredeti gráfnak. A felhasználó ki tudja bontani és össze tudja csukni az egyes csomópontokat, így a fontosakat láthatóvá lehet tenni, a kevésbé fontosakat pedig el lehet rejteni. A csomópontok kibontása és összecsukása azt jelenti, hogy a szomszédokat láthatjuk vagy nem. Ez a módszer hasonló a Dokulil-féle megoldáshoz [59], csak itt a csomópontok és élek nem csak címkéket tartalmaznak, hanem összetettebb struktúrát, mint például a típusait a csomópontoknak és az éleknek. Ez a megközelítés több információt ad, de cserébe a sok információ miatt kisebb részt láthatunk a gráfból egyszerre. A LODmilla hasonlóan működik, de csak a kiválasztott csomópontról kapunk leírást vagy képet. Ez a megjelenítő kifejezetten a szemantikus adatok megjelenítésére készült és forrásként a LOD felhőt használja. A Vizboard rendszer egy folyamatot ad, amely lehetőséget ad a felhasználónak, hogy megjelenítse és elemezze a szemantikus adathalmazt anélkül, hogy lenne szakértői ismerete vagy programozói tudása. A folyamat öt lépésből tevődik össze. Az első lépés az adatok feltöltése, amelyből a rendszer már alap statisztikákat készít, amit a felhasználó használni tud. Ezután a rendszer klaszterezést alkalmaz, ami részgráfokat alkot és a felhasználó ki tudja választani a számára fontosat. Ezután vizualizációs komponenseket ajánl fel, amiknek a beállításait finomítani lehet. Ezután a rendszer megjeleníti az adatokat, ahol a rejtett tudások megjelenhetnek. A folyamat hasonló a mi rendszerünk működéséhez, ahol első lépésként meg kell adni az adathalmazt, majd kiválasztani

a fontosabb tulajdonságokat amiket *lánc* vagy *nem-lánc*-ként kezel a rendszer, majd az eredményt olyan formában készítjük el, amely már megjeleníthető egy másik szoftverrel.

A keletkező adatok feldolgozása kihívást jelent a méretük miatt. Az elosztott számítási modellek megoldást jelentenek erre a problémára. Ezek közé tartozik a MapReduce modell [58] is, amelyet a prototípusnál használunk. Schätzle a cikkében [110] MapReduce-szal történő biszimulációt [104] mutat be. A különbség az ők és a mi megoldásunk között, hogy ők iteratív algoritmust alkalmaznak, amelynek a futási ideje több, mint az általunk bemutatott megoldás.

A MapReduce modell előnyeit többen próbálták alkalmazni a szemantikus adatokra is. A legelső eredmények Hadoop [98] rendszeren történő kiértékelések voltak. A legelső ilyen eredményeket Husain és társai [71, 75] mutatták be. Cikkükben az adatok optimális tárolásával és a kiértékeléssel is foglalkoztak. Ezután jött több megoldás is, amik szintén a Hadoop-ot használták. Ilyen rendszerek például a h2rdf [96] vagy a hadoopsparql [95]. A Hadoop után megjelentek különböző keretrendszerek, amelyek a Hadoopra épültek, de más programozási nyelvvel rendelekeztek, mint például a Pig, amelynek saját lekérdező nyelve van a Pig Latin, melyre Schätzle és társai implementálták a Pigsparql-t [89], ami a szemantikus adatok lekérdezését tette lehetővé. Egy másik rendszer az Apache Impala, ami egy elosztott SQL lekérdező motor a Hadoophoz. Szintén Schätzle és társai implementálták a lekérdező nyelvet erre a keretrendszerre Sempala [117] néven.

A Hadoop rendszer problémája, hogy túl sok I/O műveletet használ, emiatt lassú. A Spark egy új keretrendszer, ami egy memória alapú osztott számítást tesz lehetővé. Ezzel indult el a szemantikus adatok kiértékelésének irányzata a Sparkon. Első ilyen eredményekről olvashatunk Abdelaziz és társai cikkében, ami Spartex [119] vagy a Schätzle és társai által publikált cikkben [121]. A Spark keretrendszert kiegészítették több könyvtárral is, amelyek különböző adatok feldolgozására szolgálnak. Az egyik ilyen könyvtár a GraphX, amely gráfok osztott feldolgozására szolgál. Mivel a szemantikus adatok felfoghatóak gráfként is, így indult el a 'szemantikus adatok lekérdezése osztott gráf elemző rendszereken' kutatási irány. Ebben a témában jelenleg kevés eredmény található. Az egyik a Goodman és társai [115] által bemutatott algoritmus, amely GraphLabon működik. Ez a megoldás hasonlít a megoldásomra, hisz a szerzők szintén a csúcs attribútumaként tárolták a Data property-ket valamint a csúcsok tárolják a részeredményeket. A különbség az ő megoldásukkal szemben, hogy nem minden lekérdezésre képes eredményt adni. Egy másik megoldás a Spark GraphX-en futó S2X [122], amelyet Schätzle és társai készítettek. Ebben a megoldásban minden iterációban minden csomópont végez számítást, ellenben a mi megoldásunknál az egyes iterációkban csak az adott típusú élek küldhetnek üzenetet.



5.1. ábra. Csúcs összevonás példa. Olyan csúcsokat tudunk összevonni biszimulációval, akik csak *nem-lánc* tulajdonsággal rendelkeznek és azonosak az állítmányaik. Az összevont csúcsok új címkéje az összevont csúcsok számossága lesz.

5.3. Szemantikus adatok elemzése Hadoopon [7]

A szemantikus adatok terjedésének és összekapcsolhatóságuknak köszönhetően, az adatok mérete és komplexitása olyan nagy lehet, amit már nem tudunk kezelni egyszerű alkalmazásokkal. Emiatt van szükség új eszközökre, amelyek ezeket az adatokat tudják kezelni. Ebben a fejezetben bemutatok egy olyan algoritmust, amely Big Data eszközöket használ arra, hogy a szemantikus gráfot redukálja olyan szinten, amit már meg tudnak jeleníteni a gráf megjelenítő eszközök. Ezek a memóriában tárolják az adatokat, emiatt ha nagy a gráfunk, akkor nem tudják megjeleníteni azt. Az algoritmus két részből áll: az első rész a szűrésre szolgál és csak azokat az éleket hagyja benne a gráfban, amelyeket meg szeretnénk jeleníteni a felületen. A második fázis pedig redukálja azokat az éleket, amelyeket aggregálva szeretnénk látni. Az algoritmus elkészítéséhez a MapReduce modellt használtuk, amely nagy adatok kezelésére lett kifejlesztve. A probléma felvetés és a formális leírás Rácz Gábor érdeme, még az algoritmus és annak implementálása Hadoop környezetbe saját eredmény.

5.3.1. Algoritmusok

Szűrés

A 20. algoritmus az első fázisban futó Job kódját mutatja be. Ez a fázis készíti elő a hármasokat az összevonáshoz. A Map fázis beolvassa az N-Triples formátumú fájlokat soronként, majd a sorokat szétbontja alany, állítmány és tárgy részekre (2. sor). Ha az

állítmányt kiválasztottuk *lánc* vagy *nem-lánc* tulajdonságnak, akkor egy kulcs-érték pár készül, amelyet a Reduce fázis fog feldolgozni. A párnak a kulcsa a tárgy, az értéke pedig az állítmányból és az alanyból készített pár (4. sor). Ha olyan állítmányt olvas, amely nem szükséges az eredményhez, akkor azt figyelmen kívül hagyja az algoritmus, ezzel csökkentve a gráf méretét. Az így elkészült kulcs-érték párokat a reducer folyamatok fogják megkapni. Minden egyes reducer folyamat megkap egy kulcsot és az ahhoz tartozó összes értéket. Ezeket az értékeket rendezzük az állítmányok alapján, majd az eredményt kiírjuk egy fájlba. Ezt az eredményt fogja feldolgozni a második algoritmus.

Algoritmus 20 Szűrés

```
 1: procedure MAP(line)                          ▷ line alany-állítmány-tárgy formátumú
 2:     (s, p, o) = split(line)
 3:     if p is predefined then
 4:         emit(s, (p, o))
 5:     end if
 6: end procedure

    (* key az alany, values a (állítmány, tárgy) párok listája *)
 7: procedure REDUCE(key, values)
 8:     sort(values)                             ▷ rendezett (p, o) párok a p alapján
 9:     emit(values, key)
10: end procedure
```

Összevonás

A második fázis, amit a 21. algoritmus ír le, az összevonás, ami a hasonló csúcsok összevonását jelenti. Két csúcsot akkor nevezünk hasonlónak, ha ugyanolyan állítmánnyal rendelkező kimenő élekkel rendelkeznek, ahogy azt a 8. definíció leírja. A Map fázis beolvassa az előző algoritmus kimeneteként létrejött fájlt soronként és kulcs-érték párokat készít, amelyben az érték az alany, a kulcs pedig az állítmány és tárgy párosa. A hasonlóságot úgy ellenőrizzük, hogy az állítmány-tárgy páros kulcsot rendezve hozzuk létre. Ezzel a megoldással minden ugyanolyan élekkel rendelkező csúcs egy kulcshoz fog kerülni. A Reducer folyamat megkapja ezeket a csúcsokat, majd eldönti, hogy össze kell-e vonni őket. Csúcsot akkor tudunk összevonni, ha a csúcs nem tartalmaz olyan kimenő élt, amely *lánc*-ként van megjelölve. Ha tartalmaz, akkor a csúcs a gráf struktúráját reprezentálja, ellenkező esetben a csúcs típus számosságát.

A 5.1. ábrán egy kis példán mutatom be, hogyan történik két csúcs összevonása. A bal oldalon láthatjuk a bemeneti gráfot, amelynek két típusú éle van. A szaggatott él meg van jelölve *nem-lánc* tulajdonságnak, még a folytonos élek a *lánc* tulajdonságok.

Ahogy a jobb oldalon láthatjuk az alsó két csúcs lett összevonva, mivel ezek a csúcsok rendelkeznek ugyanolyan *nem-lánc* élekkel. A felső csúcs azért nem lett hozzávéve, mert az csak a *dbpedia:Hungary* csúcshoz csatlakozik. Láthatjuk azt is, hogy az összevont csúcs címkéje 2-re változott.

Algoritmus 21 Összevonás

```
1: procedure MAP(line)
2:     (< (p_1, o_1), (p_2, o_2), ...(p_n, o_n) >, s) = split(line)
3:     emit(< (p_1, o_1), (p_2, o_2), ...(p_n, o_n) >, s)
4: end procedure

   (* key rendezett listája a (állítmány, tárgy) pároknak, values alanyok listája, amely tartalmazza a párokat *)
5: procedure REDUCE(key, values)
6:     p_list = getPredicatesFromKey(key)
7:     if p_list nem tartalmazz olyan állítmányt ami meg van jelölve lánc-ként then
8:         for all (p, o) ∈ key do
9:             emit(count(values), p, o)
10:        end for
11:    else
12:        for all s ∈ values do
13:            for all (p, o) ∈ key do
14:                emit(s, p, o)
15:            end for
16:        end for
17:    end if
18: end procedure
```

5.3.2. Kiértékelés

Teljesítmény

A módszerünket két adathalmazon teszteltük: a DBpedia 3.8[4] és a Freebase[5] adatain. A DBpedia a Wikipediából kinyert adatokat tartalmazza, még a Freebase egy közösségi tudástár, amelyet a tagok kezelnek. Mindkét adathalmaz elérhető N-Triples formátumban, amely lehetőséget ad arra, hogy soronként dolgozzuk fel a hármasokat. A DBpedia közel 40 millió hármas tartalmazott és 5.4 GB méretű volt. A Freebase adathalmaz nagyobb 1.9 milliárd hármassal és 250 GB mérettel rendelkezett.

A Hadoop klaszter 13 gépet tartalmazott, és minden gép Intel Core i5-650-es 2*3.2 GHz processzorral és 4 GB memóriával valamint 150 GB tárhellyel rendelkezett a HDFS szá-

[4] http://wiki.dbpedia.org/Downloads38

[5] http://download.freebaseapps.com

5.1. táblázat. A futásidők és az eredmény sorok száma a különböző konfigurációkkal a DBpedia és Freebase adathalmazokon

Adathalmaz	**Konfiguráció**	**Futásidő**	**Sorok száma**
DBpedia	rdfs:subClassOf (c) rdf:type (nc)	195s	2,212
	dbpo:locatedInArea (c)	80s	32,517
	dbpo:locatedInArea (c) dbpo:birthPlace (nc)	88s	194,806
Freebase	freebase:religion.religion.branched_from (c) freebase:people.person.religion (nc)	863s	2,382

mára. Minden konfigurációt hatszor futtattunk le, ahol az első futás eredményét nem vettük figyelembe, majd a maradék öt futás átlagát vettük. A 5.1. táblázatban láthatjuk az eredményeket a különböző adathalmazokon és különböző konfigurációval végzett futásoknak. A táblázat tartalmazza a futási időt és az eredmény méretét is, amit a szűrés és összevonás fázisok után kaptunk. Meg kell jegyeznünk, hogy a Hadoopnak van egy alap idő költsége, hogy elindítson egy Job futást. Ennek az adminisztrációs költsége 30 másodperc, ami a mi esetünkben a két Job inditás miatt 60 másodperc volt.

Az első konfiguráció a DBpedia osztályhierarchia vizsgálatára szolgál. Az eredményül kapott gráfon láthatjuk az osztályok közötti alárendeltséget és az egyes osztályokhoz tartozó egyedek számosságát. Az *rdfs:subClassOf* állítmány jelenti, hogy egy osztály alosztálya egy másiknak. Ezt az állítmány jelöltük meg *lánc* tulajdonságnak. Az *rdf:type* állítmány pedig az objektumok típusát mondja meg, amelyet *nem-lánc*-ként jelöltünk meg, így megkapva a számosságot. A következő konfiguráció a geográfiai kapcsolatokat reprezentálja különböző területeknek. Itt a *dbpedia:locatedInArea* tulajdonságot választottuk *lánc*-nak mert a különböző területek hierarchikusan kapcsolatban vannak egymással. *Nem-lánc* tulajdonságnak pedig a *dbpedia:birthPlace* állítmányt alkalmaztuk, amely az egyes entitásokat a születési helyükkel kapcsol össze. Ezzel a konfigurációval láthatjuk, hogy az adathalmazban szereplő entitások közül hol, mennyien születtek. A Freebase adathalmazból a vallás hierarchiát jelenítettük meg. Ehhez a *freebase:religion.religion.branched_from* állítmányt használtuk. Ez az állítmány azt mondja meg, hogy egy vallás melyik vallásból származik. Ezenkívül azt is szerettük volna még megjeleníteni, hogy az egyes vallásokhoz hány személy kapcsolódik, emiatt a *freebase:people.person.religion* tulajdonságot jelöltük meg *nem-lánc*-ként.

A 5.1. táblázatban láthatjuk, hogy az első eset futási ideje két és félszer több, mint a második és harmadik eseté, pedig mind a DBpediát használták inputként. Ennek oka, hogy az *rdf:type* tulajdonság a legtöbbet előforduló tulajdonság az adathalmazban, emiatt az összevonás algoritmusnak több csomópontot kellett összevonnia, ami időigényes.

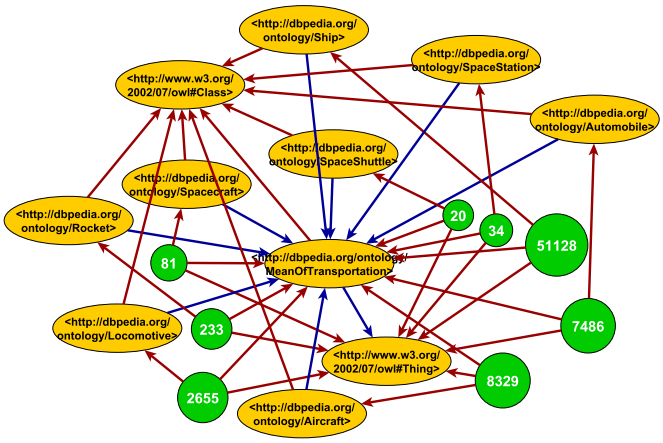
Másfelől azt is láthatjuk, hogy nagyobb adathalmazra is, mint a Freebase, a módszerünk hatékonyan működik.

Eredményül kapott gráfok vizsgálata

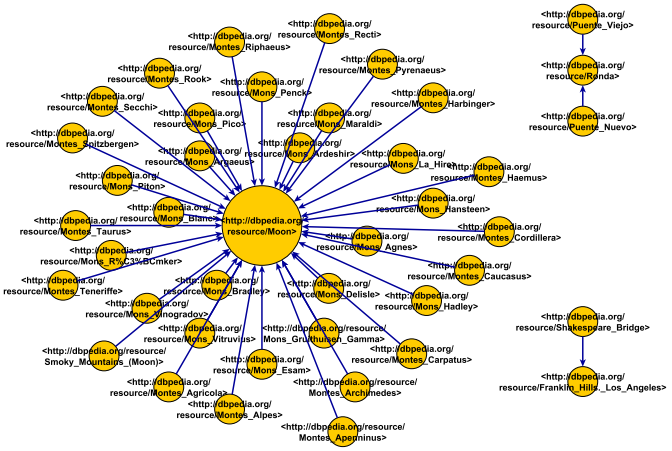
Miután az alkalmazásunkat lefuttattuk a fenti konfigurációkkal, az eredményt megjelenítettük a yEd gráf megjelenítő és szerkesztő eszközzel. A 5.2. ábrán a DBpediából kapott gráfok részleteit láthatjuk. A 5.2(a). ábra a közlekedési eszközök közötti kapcsolatot mutatja meg és hogy az egyes jármű osztályhoz hány entitás tartozik. Az egyes entitások tartozhatnak akár több osztályhoz is. Az ábrán a kék élek jelentik a *lánc* tulajdonságot még a pirosak a *nem-lánc* kapcsolatokat. Ez az ábra jól láthatóan szemlélteti az eredményünket egy kisebb gráfon. A 5.2(c). ábrán a második konfiguráció eredményét láthatjuk, ahol a geográfiai kapcsolatok láthatóak a *dbpedia:locatedInArea* tulajdonság segítségével. Ahogy várható volt az eredményül kapott gráf több nagyobb komponensből állt, amik főként a kontinenseket és az ott található országokat tartalmazta. Az elvárásainkkal szemben csak két nagyobb gráfot kaptunk, amelyek Európát és Amerikát reprezentálják. Ezenkívül kaptunk még pár közepes méretű és sok apró, akár két csúcsból álló gráfot is, amelyek függetlenek voltak. A közepes méretű gráfok főként szigetek, mint például Grönland vagy a Húsvét-szigetek, de köztük volt a Hold is a kráterreivel. A kis méretű gráfok viszont hibák az adathalmazban, amelyek rossz helyre kapcsolódnak, vagy épp nem kapcsolódnak. Megvizsgálva ezeket, olyan hibákat kaptunk, hogy például a hidak rosszul vannak az adathalmazban, hisz azok két városhoz is tartozhatnak, de nem kapcsolattal vannak ezek kifejezve, hanem az entitás azonosítójában. Ilyen például: *http://dbpedia.org/resource/Franklin_Hills,_Los_Angeles*. Franklin Hills egy városrész Los Angelesben, de ez nem kapcsolódik Los Angeleshez, viszont az ott található híd ehhez az entitáshoz kapcsolódik. Ezeket láthatjuk a 5.2(b). ábrán. Itt szerepel még példának a Ronda spanyol provincia, amely szintén nem szerepel megfelelő kapcsolatokkal az adathalmazban illetve a Hold a krátereivel.

A 5.3. ábra az utolsó konfigurációk eredményét mutatja be, ahol a *freebase:branched_from* kapcsolatot vizsgáltuk a különböző vallások között és a számosságukat a Freebase adathalmazban követőik alapján. A Freebase adathalmazban található vallások száma korlátos, de több ember is található benne, aki több vallást is követ egyszerre. Ez a nagy komponens látható az ábra bal oldalán. Ennél a konfigurációnál is megjelentek kisebb komponensek, amelyek nem kapcsolódnak a *freebase:branched_from* állítmánnyal más valláshoz, mint például a Thelema[6] vallás. Másfelől itt is találhatunk

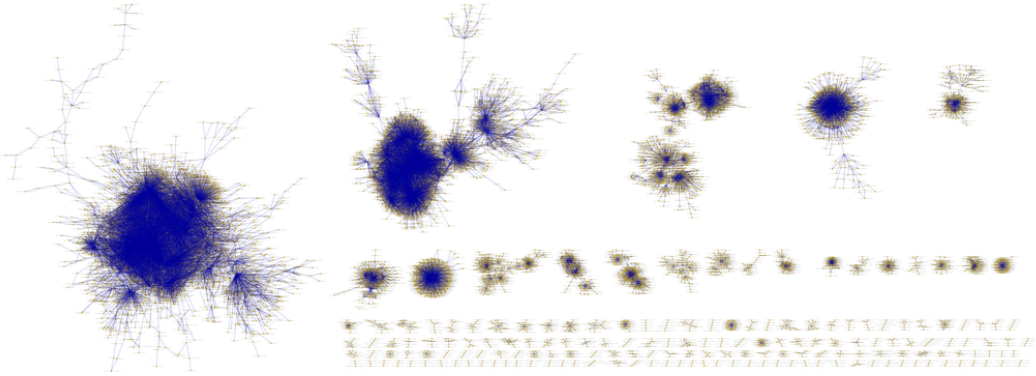
[6] www.freebase.com/m/07grj



(a) Osztályhierarchia a közlekedési eszközök között és azok számossága



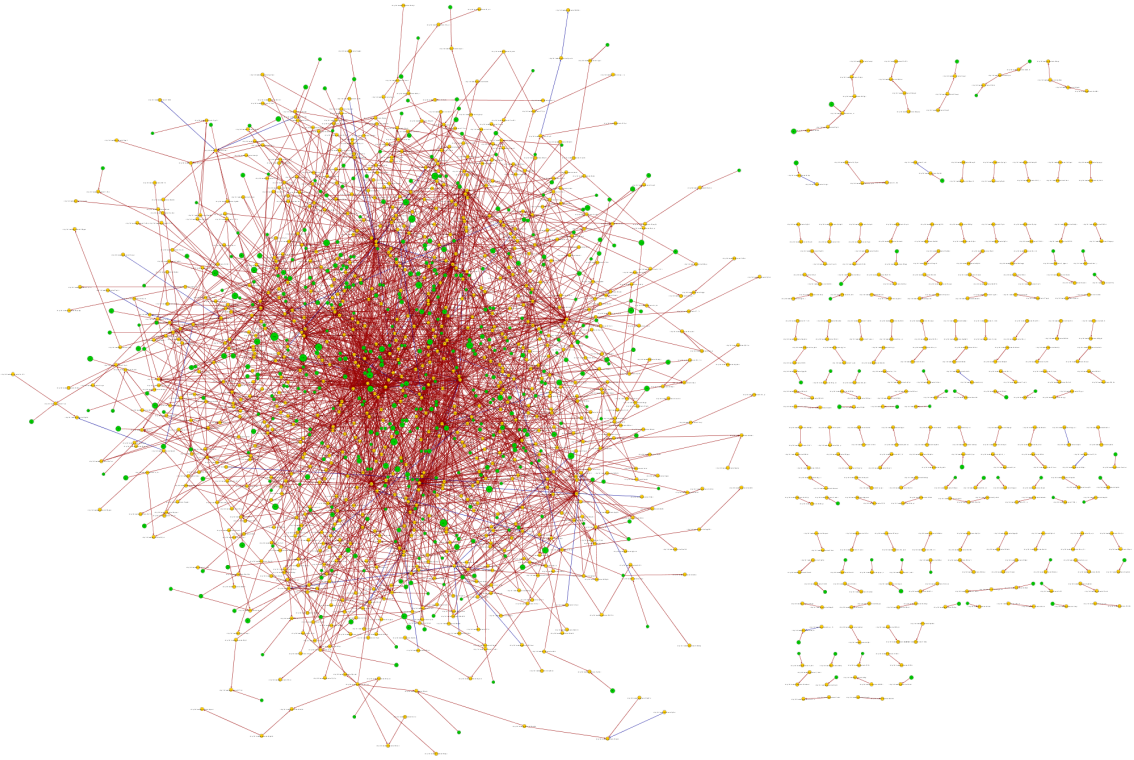
(b) A hold és a kráterei (bal) Ronda és Franklin Hills (jobb)



(c) Geográfiai kapcsolatok a DBpediában a dbpedia:locatedInArea tulajdonsággal

5.2. ábra. Gráf részletek a DBpedia adathalmazból. Az ábrák a DBpedián futtatott eredményeket szemlélteti. Az a) ábrán a *lánc* tulajdonság *rdfs:subClassOf* a *nem-lánc* pedig a *rdf:type* tulajdonság. A b) és c) ábrán *lánc* tulajdonság-ként a *dbpedia:locatedInArea* tulajdonságot alkalmaztuk és nem volt *nem-lánc* tulajdonság. Az a) ábra egy várt gráfot mutat, még a b) ábrán olyan példákat látunk, amelyek hibák, érdekességek az adathalmazban. A Franklin Hill nem kapcsolódik Los Angeleshez. A c) ábrán a teljes DBpedia eredmény látható.

olyan állításokat, amelyek hibásan szerepelnek az adathalmazban. Például James Zabiela[7] vallása Jedi[8], pedig a Jedi nem vallásként szerepel az adathalmazban, hanem művészként. Egy másik példa Matthew Russell[9] vallása látványtervező[10], ami egy foglalkozás. Ezek a példák jól szemléltetik az algoritmus hasznosságát, a hibák feltárására és az adathalmaz megismerésére.



5.3. ábra. Gráf a Freebase adathalmazból. Az ábra a gráf szétesését szemlélteti. *Lánc* tulajdonságnak a *freebase:branched_from*, *nem-lánc* tulajdonságnak pedig a *freebase:person:religion*. Ez a vallások kapcsolatait és a követőinek számosságát szemlélteti. Jobb oldalt látjuk a gráf hibáit.

5.4. Spar(k)ql: SPARQL lekérdezések kiértékelése Spark GraphX rendszeren [9]

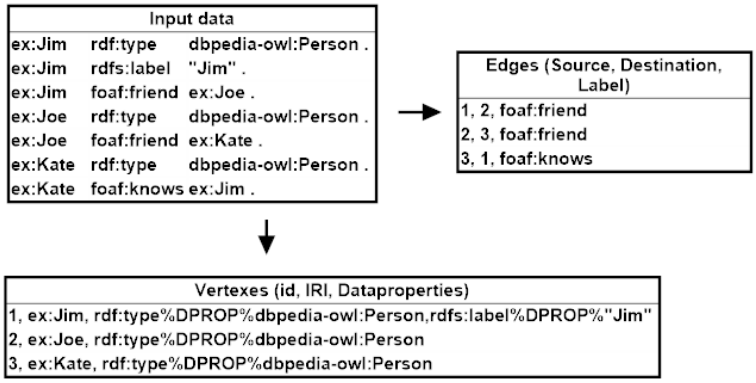
Egy másik kutatási lehetőség a szemantikus lekérdezések megválaszolása Big Data eszközökkel. A korábbi munkákban láthattuk több kutató is foglalkozik a témával. Ebben a fejezetben bemutatok egy algoritmust, amely egy megoldást jelent az osztott gráf elemző

[7] www.freebase.com/m/01rh56d
[8] www.freebase.com/m/01wtsmx
[9] www.freebase.com/m/0plk41m
[10] http://www.freebase.com/m/02pjxr

rendszereken történő kiértékelésben. Majd bemutatom ennek a megoldásnak a javított változatát is. Az itt bemutatott algoritmusok önálló eredmények.

5.4.1. Gráf betöltés

Az osztott gráf feldolgozás első lépése a gráf betöltése a rendszerbe. A Spark RDD-kben tárolja a gráfot, amely a memóriában helyezkedik el. A szemantikus gráfok betöltése úgy történik, hogy beolvasunk egy hármast, amiből elkészítjük a csúcsokat és az élt. Ha létezik az adott csúcs akkor azt meg kell keresni a már betöltött adatok között. A betöltés során arra is figyelni kell, hogy az adott él Object property vagy Data property, azaz a tárgy helyén egy másik csúcs áll, vagy egy konkrét érték. Ha Data property tulajdonságról van szó, akkor az nem egy új él lesz, hanem a csúcs egy tulajdonsága. Ahhoz, hogy gyorsítani tudjuk a gráf betöltését elkészítettem egy olyan formátumot, amely egy sorban tárolja a csúcsot és a hozzá tartozó Data property-ket, illetve a csúcsok már rendelkeznek azonosítóval. Egy másik fájlban pedig tároljuk az éleket a csúcsok között. Ezt láthatjuk a 5.4. ábrán. Ez az átalakítás a lekérdezést feldolgozó algoritmus előtt történt meg. A kiértékelésbe nem számolom bele ennek az ideje.



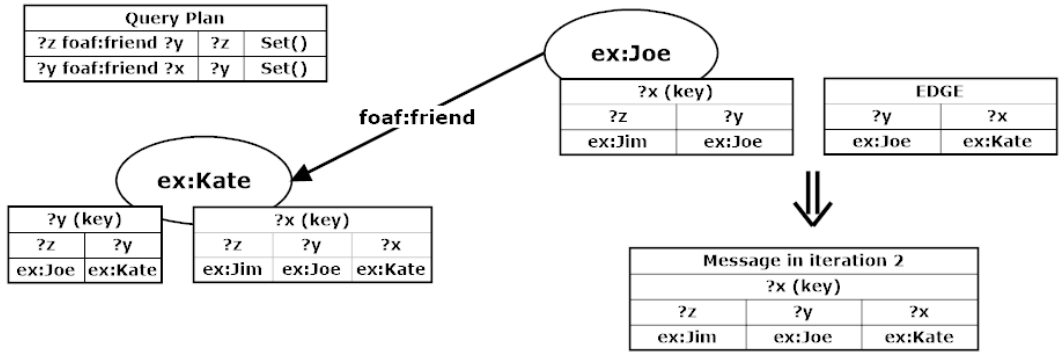
5.4. ábra. Gráf betöltés

5.4.2. Csúcs és Üzenet modell

Ahogy említettem az osztott gráf elemző rendszerek úgy működnek, hogy a csúcsok tárolnak információkat és az üzenetek alapján frissítik azokat. A betöltött gráf csúcsai az esetemben a Data Property-ket tárolják, valamint egyes Object property-ket. Goodman és társai [115] csak az $rdf:type$ és $rdfs:label$ tulajdonságokat tárolták. Én szintén tárolom ezeket az értékeket a csúcson, aminek az oka, hogy egy lekérdezésben ez az állítmány

gyakran szerepel. A csúcsok ezenkívül tárolják a saját URI-jukat, valamint a kiértékelés részeredményét.

Az eredmények egy HashMap-ként vannak tárolva, ahol a kulcs egy SPARQL változó, amelyet az adott csúcs helyettesít. Az érték pedig egy táblázat, amelyben szerepel az összes eddigi részeredmény. Egy csúcs akár több változót is helyettesíthet a kiértékelés során. Erre láthatunk példát a 5.5. ábrán, ahol az *ex:Kate* az *?x* és *?y* értéket is felvette.



5.5. ábra. Csúcs és üzenet modell. A *ex:Joe* csomópont küldi el a *?x* változóhoz tartozó táblázatát, amit összekapcsol előtte az éllel. Ezt az üzenetet fogja a *ex:Kate* csomópont összekapcsolni a saját *?y* táblájával.

A Spark a gráfot úgy elemzi, hogy minden iterációban végig megy azokon az éleken amelyek aktívak. Egy él akkor aktív, ha az előző iterációban valamelyik csúcsa kapott üzenetet. Itt a probléma a következő: vegyünk egy lekérdezés:t ?x foaf:friend ?y . ?y foaf:knows ?z . ?w foaf:friend ?v . ?v foaf:knows ?z, és legyen a lekérdezési terv olyan, hogy egyszerűen végig megy a hármasmintákon. Ebben az esetben láthatjuk, hogy az első iterációban az első feltétel miatt az *?y* kap üzenetet, emiatt a második él aktív lesz, mivel van olyan csúcsa, amely kapott üzenetet. A második iterációban a *?z* kap üzenetet, emiatt a negyedik él lesz aktív. A következő iterációban olyan élek kellenek amelyek aktívak és a címkéjük $foaf : friend$. Mivel az előző körben a ?z változót helyettesítő csúcsok kaptak üzenetet, így nekik vannak aktív éleik. A ?w és ?v csúcsoknak nem lesz aktív élei, így a kiértékelés megáll és nem ad eredményt. Ahhoz, hogy minden él aktív maradjon mindaddig amíg szükség van rájuk, bevezettem egy *alive* nevezetű üzenet típust. Ez az üzenet aktív állapotban tartja csúcs éleit. Az *alive* üzenetek minden olyan élen mennek, amelyek később lesznek kiértékelve. Fontos megjegyezni, hogy a GraphX indításánál meg kell adni egy kezdeti üzenetet, amit minden csúcs meg fog kapni. Ezáltal kezdetben minden él aktív lesz a legelső iterációban.

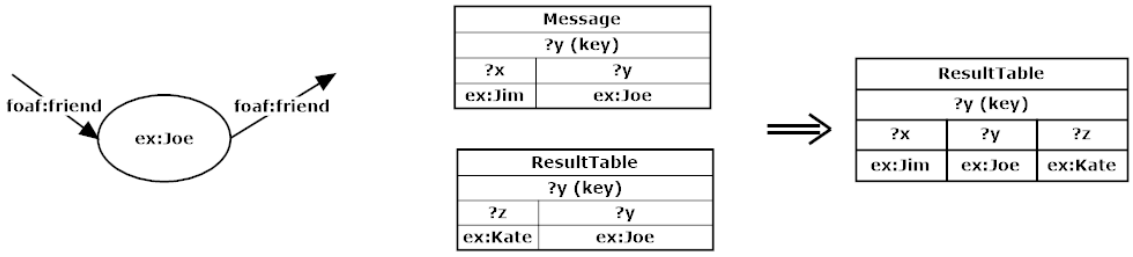
A másik típusú üzenet maga a kiértékeléshez tartozó üzenet. Ebben az üzenetben egy változónév szerepel és egy eredmény tábla. A változónév megmondja, hogy a címzett

csúcs mely változót helyettesíti az adott iterációban. A táblázat pedig megadja az eddigi részeredményeket. A táblázat fejlécében szerepelnek a változók, a törzsében pedig a lehetséges értékek. Az üzenet küldés előtt a küldő csúcs összefűzi az adott élt a táblázattal és így küldi el azt a címzettnek. Az osztott gráf elemzés úgy működik, hogy a cél csúcs csak egy üzenetet kaphat meg. Emiatt az aszinkron módon küldött üzeneteket össze kell kapcsolnunk, és az összekapcsolás módját mi írjuk le.

Az üzenetek összekapcsolásával a következő probléma lép fel. Hogyan értelmezzük két üzenet összekapcsolását. Lehetséges megoldás az unió és a join. Vannak olyan üzenetek amelyek külön eredményt jelentenek, ekkor azok uniója lesz a végső eredmény (5.7. ábra), és vannak olyanok amelyekhez hozzá kell kapcsolni a másik üzenetet, ekkor join-t kell alkalmaznunk (5.6. ábra). Lássunk erre is pár példát.

Vegyünk egy lekérdezést (?x dbpedia-owl:birthPlace ?y). Ekkor az adatgráfban található városok és országok lesznek az ?y helyen. A megoldásban szerepelni fog minden olyan ?x, aki ugyanazon a helyen született. Például, ha a hely *dbpedia:Budapest*, akkor ez a csúcs fogja megkapni az üzeneteket azoktól a csúcsoktól, amelyeknek van *dbpedia-owl:birthPlace* éle. Ebben az esetben unióra van szükségünk.

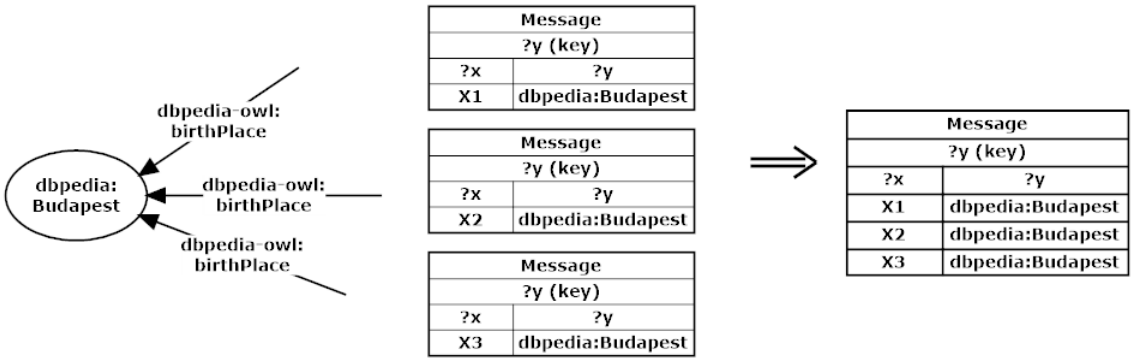
Vegyük a következő lekérdezést (?x foaf:friend ?y . ?z foaf:knows ?y), ekkor mind a két iterációban a ?y helyen lévő csúcsok fognak üzenetet kapni. Az első iterációban az ?x és az ?y kap értéket és ez fog szerepelni az üzenetben. A következő iterációban viszont már van korábbi üzenet az ?y-nál, amelynek a fejlécei mások. Ekkor joinolni kell az adatokat, az alapján, hogy minden közös változó helyén azonos érték szerepel. Tehát akkor alkalmazunk két üzenetre uniót, ha a fejléceik megegyeznek. Ezek a megoldások érvényesek a csúcson történő üzenet összekapcsoláskor is.



5.6. ábra. Üzenet és részeredmény összekapcsolása

5.4.3. Lekérdezési terv készítése

A SPARQL lekérdezés kiértékeléséhez meg kell határoznunk egy sorrendet a hármasminták között. Ez a sorrend lesz az üzenetek küldésének sorrendje. Az osztott gráf kiértékelés



5.7. ábra. Üzenetek uniózása. Az egy csomóponthoz érkező üzenetek azonos sémájúak, emiatt ezeket össze tudjuk uniózni és egy üzenetként továbbítani azokat.

minden iterációjában egy hármasminta alapján lesz elküldve az üzenet. A sorrend meghatározásához egy lekérdezési fát készítettem, amelyben egy adott változó a gyökér elem és minden hármasminta egy-egy ág a fában. A fa csúcsai az alany és tárgy változók a hármasmintákból, az élek pedig a köztük lévő állítmány. A fa elkészítése lényegében egy szélességi bejárás. Az algoritmus (22. algoritmus) első lépésként felosztja a hármasmintákat Data és Object property-kre. Erre azért van szükség, mert a modellemnél a Data property-k a csúcsokon vannak tárolva, így azokat nem kell üzenetként elküldeni. Emiatt csak az Object property-kből készítjük el a fát. Ha a lekérdezés nem tartalmaz Object property-t, akkor a lekérdezési terv kész van, és csak azok a csúcsok lesznek az eredmények, amelyekre teljesülnek a Data property feltételek. Ellenkező esetben a változót berakjuk egy sorba, amiben tároljuk a még fel nem dolgozott változókat. Egy másik halmazban (*vars*) pedig tároljuk a változókat, amiknek szerepelniük kell a végső megoldásban. Amikor a gráf feldolgozás iterációs lépései véget érnek, akkor csak azok a csúcsok lesznek az eredmények, ahol a csúcs rendelkezik a gyökér elemnek választott változóval és a részeredménye rendelkezik az összes változóval, ami a *vars* halmazban van.

A fa elkészítése a következő: veszünk egy változót a sorból, és vesszük az összes olyan hármasmintát, amelyben szerepel ez a változó és még nem volt feldolgozva. Ekkor ezeket a hármasmintákat feldogozzuk, a változóit berakjuk a sorba, ha még nem szerepeltek benne és berakjuk őket a *vars* halmazba is. Ezután elkészítünk egy *PlanItem*-et, amik tárolják a hármasmintát és az üzenetküldés irányát. Az üzenet küldés mindig a gyerektől a szülő felé történik, de egy hármasmintánál ez lehet a tárgy vagy az alany változó is. Ezután az élt feldolgozottnak állítjuk be. Fontos megjegyezni, hogy ebben a megoldásban egy iterációban csak egy hármasminta lesz feldolgozva. Ennek oka, hogy az üzenetek kezelése ekkor egyszerűbb, mert egy sémájú üzenetek érkeznek csak.

Ha elkészítettük az átmeneti sorrendet, akkor minden PlanItem-hez hozzávesszük az az elfogadható fejléceket. Ez a feltétel azt határozza meg, hogy milyen feltételnek kell teljesülnie az adott üzenetre, hogy az elküldhető legyen. Ha az adott iterációban lévő élen szereplő forrás csúcs nem tartalmazza ezeket a feltételeket, akkor az ő részeredménye nem fog szerepelni a végeredményben. Ha a korábbi iterációkban a forráscsúcs kapott megfelelő üzenetet, akkor minden változó vagy konstans szerepelni fog az ő eredmény táblájában. Ezzel a feltétellel tudjuk csökkenteni az üzenetek számát. A feltételek elkészítését írja le az 23. algoritmus. Első lépésként megfordítja a lekérdezési sorrendet, mivel a fa felépítése a gyökérből indult ki, de a lekérdezések a levelektől indulnak majd. Ezután létrehoz egy *headers* halmazt, amely tartalmazni fogja a korábbi lépésben keletkezett oszlop változókat. Első lépésként összeuniózzuk a forrás és a cél változóhoz tartozó oszlop információkat, melyek kezdetben üresek, majd hozzávesszük a *src* változót a *dest* halmazhoz. Végül beállítjuk az *acceptHeader* információt az adott *PlanItem*hez.

Végső lépésként pedig elkészítjük a GraphX-hez szükséges 'alive' üzeneteket, amik ahhoz kellenek, hogy aktívan tartsuk az éleket addig, amíg szükség lesz rájuk. A GraphX az aktív éleken megy végig és az ott található csúcsokkal tudjuk elkészíteni az üzeneteket. Egy adott iteráció 'alive' üzeneteinek elkészítéséhez a későbbi iterációk hármasmintáira van szükség. Minden iterációban annyi 'alive' típusú üzenetet készítünk, ahány iteráció van még hátra a kiértékelésből. Az 'alive' üzenet egy olyan hármasminta, ahol az állítmány a későbbi hármasminták állítmánya, az alany és a tárgy pedig egy *?alive* változó.

5.4.4. SPARQL lekérdezés kiértékelése

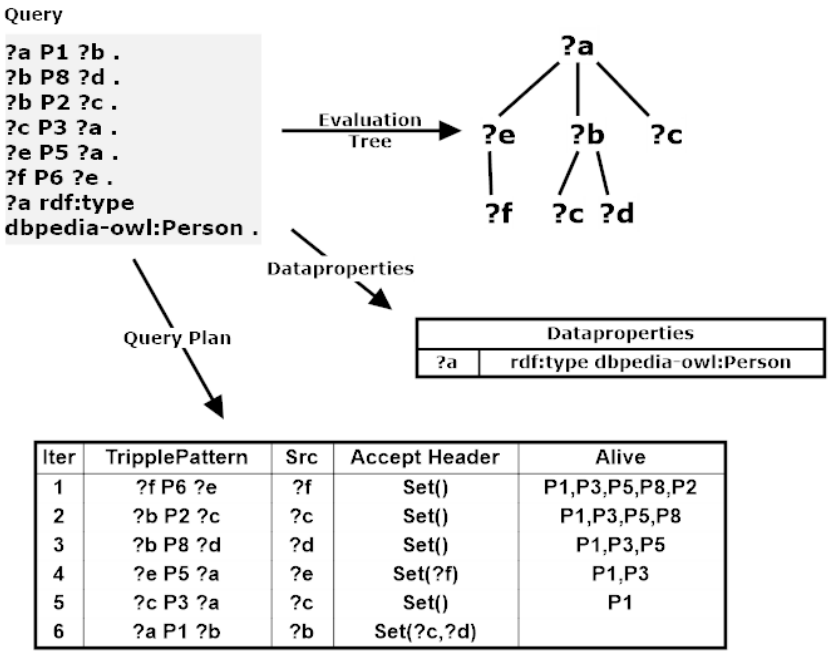
A Sparkql-ben a lekérdezés kiértékelése a következőképpen néz ki. A rendszer beolvassa a gráfot a memóriába, majd elkészíti a lekérdezési tervet. A GraphX elindításakor minden csúcs kap egy üres üzenetet, majd elindul az algoritmus. Az első iterációnál minden csúcs aktív. A GraphX végig megy az éleken és elkészíti a szükséges üzeneteket. Üzenetet készítünk akkor, amikor olyan élhez érünk, amely a lekérdezési tervben az adott iterációban szerepel. Több ilyen él is lehet, hisz az *alive* üzenetek miatt a később használt éleken is küldünk üzenetet. A lehetséges éleket tovább szűkítjük az alapján, hogy a forrás csúcs tartalmazza-e a megfelelő adattáblát. Olyan éleken nem megy üzenet, ahol a forráscsúcsban nincs meg a korábbi iterációk eredménye. További szűkítés, hogy csak olyan csúcsok küldhetnek üzenetet, akiknek megfelelő a Data property tulajdonságaik is. Olyan csúcs akinek nincs meg a megfelelő Data property-je nem lesz benne az eredményben. A kiértékelés során lesznek olyan üzenetek is, amelyek hurok éleken fognak végig menni. Ennek oka, hogy a gráf kódolásánál én minden olyan Object property-t is a csúcs

Algoritmus 22 Lineáris lekérdezési terv készítés

```
1: function createPlan(sparqlQuery)
2:     (objPropPatterns, dataPropPatterns) = splitQuery(sparqlQuery)
3:     rootNode = "", vars = {},
4:     q = Queue.init()
5:     plan = List(PlanItem).init()
6:     if objPropPatterns.size > 0 then
7:         rootNode = objPropPatterns.first().s
8:         q.push(rootNode)
9:     else
10:        rootNode = dataPropPatterns.keys.first()
11:    end if
12:    while q is not empty  do
13:        actVar = q.first()
14:        for all pattern ∈ getNotProcessedPatterns(actVar, objPropPatterns) do
15:            vars.add(pattern.s)
16:            vars.add(pattern.o)
17:            if pattern.s <> actVar then
18:                q.pushIfNotContains(pattern.o)
19:                plan.append(PlanItem(pattern, pattern.o))
20:            else
21:                q.pushIfNotContains(pattern.s)
22:                plan.append(PlanItem(pattern, pattern.s))
23:            end if
24:            setPatternToProcessed(pattern)
25:        end for
26:    end while
27:    addAcceptHeaders(plan)
28:    addAliveMsgPatterns(plan)
29:    return (plan, rootNode, vars.size,dataPropPatterns)
30: end function
```

Algoritmus 23 Elfogadható fejlécek készítése

```
1: function addAcceptHeaders(plan)
2:     plan.reverse
3:     headers.init()
4:     for all planItem ∈ plan do
5:         src = planItem.src
6:         des = getOther(planItem.pattern, src)
7:         headers(des).addAll(headers(src))
8:         headers(des).add(src)
9:         planItem.setAcceptHeaders(headers(src))
10:    end for
11: end function
```



Iter	TripplePattern	Src	Accept Header	Alive
1	?f P6 ?e	?f	Set()	P1,P3,P5,P8,P2
2	?b P2 ?c	?c	Set()	P1,P3,P5,P8
3	?b P8 ?d	?d	Set()	P1,P3,P5
4	?e P5 ?a	?e	Set(?f)	P1,P3
5	?c P3 ?a	?c	Set()	P1
6	?a P1 ?b	?b	Set(?c,?d)	

5.8. ábra. Lineáris lekérdezési terv példa. A bal felső sarokban található hármasminták alapján készített kiértékelési fa, data property-k és a lekérdezési terv.

tulajdonságának vettem, amely szöveges adatot tartalmaz. A terv generátorban viszont nem tudhatjuk, hogy mely tulajdonságok azok, amelyek szövegesek. Emiatt a hurok élen történő ellenőrzés fogja azt ellenőrizni, hogy az adott csúcsnak nincs-e olyan tulajdonsága, amelyet a csúcshoz vettünk. Ezt a folyamatot írja le a 24. algoritmus, amely előbb veszi az iterációszámot, amelyet a csúcsok tárolnak, illetve a lekérdezési tervből a Data property tulajdonságokat. Ezután végig megyünk azokon a hármasmintákon, amelyek az adott iterációhoz tartoznak. Elsőnek ellenőrizzük az él állítmányát. Ha az 'LOOP', akkor ellenőrizzük a csúcs Data property-jét és ha megfelel, akkor küldünk egy üzenetet. Az üzenet lehet egy 'alive' üzenet vagy a Data property-vel kiegészített részeredmény, amely a hurokélen megy keresztül. Ha az él nem 'LOOP' címkével rendelkezik, akkor ellenőrizzük, hogy megegyezik-e a hármasmintával. Ha igen, akkor a küldés irányának megfelelően ellenőrizzük a forrás csúcs Data property-jét és a részeredményének a változóit. Ha minden megfelelő, akkor hozzákapcsoljuk az élt a részeredményhez és elküldjük a cél csúcshoz. Amikor az algoritmus befejezi a működését, nincs több üzenet a gráfon, akkor a gyökér csúcsként megjelölt csúcsokon ellenőrizzük a változók darabszámát, a Data property-ket és összegyűjtjük a részeredményeket. Azon csúcsoknál, ahol nincs meg minden változóhoz tartozó behelyettesítési érték, azok nem érvényes eredmények.

Algoritmus 24 Üzenet készítés

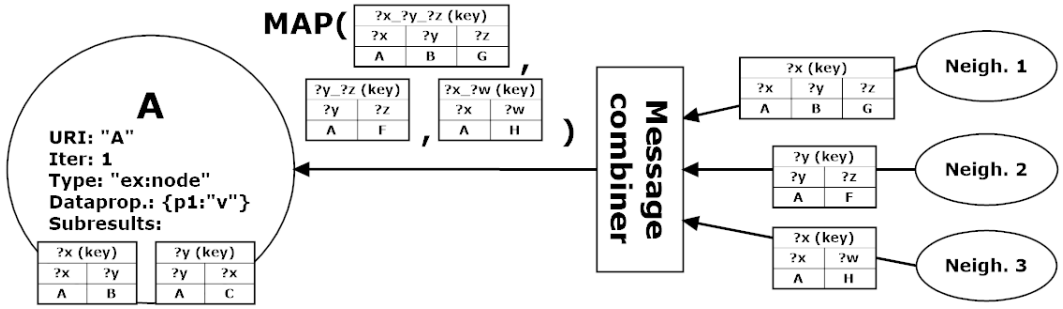
```
1: function MessageCreation(edge)
2:     iter = max(edge.src.iter, edge.dst.iter)
3:     dp = queryPlan.getDataProps()
4:     messages.init()
5:     for all tp ∈ queryPlan.plan.get(iter) do
6:         if edge.attr = "LOOP" and edge.src.checkCond(dp, tp) then
7:             if tp.isAlive then
8:                 messages.push((edge.src, initMsg))
9:             else
10:                messages.push((edge.src, merge(edge, edge.src.getSubResult(tp.s))))
11:            end if
12:        else if edge.attr = tp.p then
13:            if tp.isAlive then
14:                messages.push((edge.src, initMsg))
15:            else if edge.src = tp.src && edge.src.checkCond(dp, tp) then
16:                messages.push((edge.dst, merge(edge, edge.src.getSubResult(tp.s))))
17:            else if edge.dst.checkCond(dp, tp) then
18:                messages.push((edge.src, merge(edge, edge.dst.getSubResult(tp.o))))
19:                messages.push((edge.src, initMsg))
20:            end if
21:        end if
22:    end for
23: end function
```

5.5. P-Spar(k)ql: SPARQL kiértékelés párhuzamos lekérdezési tervvel [10]

A Sparkql egyik problémája, hogy hurok éleket használ ahhoz, hogy a Data property-ket kiértékelje. Ez felesleges élekkel és üzenetekkel jár. A másik probléma az 'alive' üzenetek, amelyek arra szolgálnak, hogy az élek aktívak maradjanak a kiértékelés során. Ez szintén felesleges üzenetekkel jár. A P-Sparkql megoldásomban bemutatom a Sparkql egy olyan javítását, ami nem használ felesleges éleket és üzeneteket.

5.5.1. Csúcs és Üzenet modell

A P-Sparkql-nél használt csúcs modell ugyanaz, mint a Sparkql-nél. Az üzenetek viszont megváltoztak. A Sparkql-ben az üzenetek egyszerű Map-ként voltak kezelve. A kulcs az adott csúcs által helyettesített változó, még az érték az aktuális részeredmény. Ahogy korábban említettem egy csúcs mindig egy üzenetként kapja meg a szomszédok által küldött összes üzenetet. A korábbi megoldás egyszerű volt, hisz egy adott iterációban csak egy típusú él küldhetett üzenetet, emiatt a forrás táblák sémája megegyezett és a



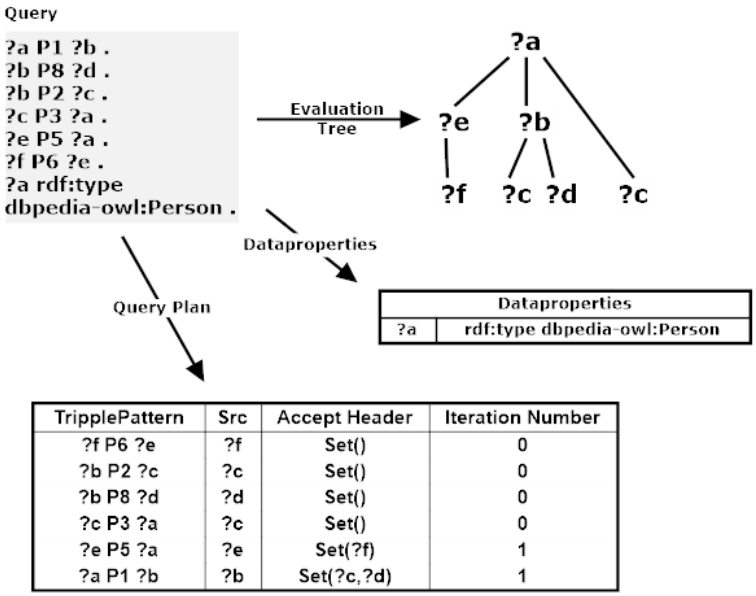
5.9. ábra. Csúcs és üzenet modell

részeredményeket csak össze kellett uniózni. Ehhez viszont kellett az, hogy minden él életben legyen mindaddig, amíg a feldolgozás eljut odáig. A párhuzamos lekérdezési terv alapján, amit láthatunk a következő fejezetben, az üzenetek különböző éleken és különböző részeredményekkel érkezhetnek. Emiatt meg kell oldanunk az üzenetek összekapcsolását. Ezt úgy oldjuk meg, hogy egy Mapben tároljuk a részeredményeket úgy, hogy a kulcs a változó és a tábla fejlécének alfabetikus sorrendje. Ekkor ha két olyan üzenet jön, aminek ez a sémája, akkor azokat egyszerűen csak összeúniózzuk. Az eredményül kapott kulcs-értékeket fogja megkapni a csúcs, aki minden kulcs értéket össze fog joinolni a saját eredményével. A Join műveletet azért nem végezhetjük el korábban, mert az üzenetek sorrendje nem determinisztikus. Emiatt lehet, hogy akkor akarunk joinolni két táblát, amikor még az adott érték nem érkezett meg a másik táblába. Példát láthatunk a 5.9. ábrán.

5.5.2. Lekérdezési terv készítése

A lekérdezési fa elkészítése hasonló, mint a Sparkql megvalósításánál. Annyi változás történik, hogy amikor a gráfot beolvassa a Spark, akkor információkat gyűjtünk a gráfról: milyen Data property-k vannak és az egyes Object property-k hányszor szerepelnek a gráfban. A lekérdezési terv ez alapján készül el, és a *splitQuery* is ez alapján tudja szétválasztani az egyes hármasmintákat. Ezzel a megoldással nincs szükségünk többé a hurok élekre, hisz tudjuk mely állítmány Data property tulajdonság. A másik információ pedig segít abban, hogy hatékonyabb lekérdezési sorrendet adjunk meg. A statisztika alapján azt az élt használjuk először, amely ritkábban szerepel a gráfban, így kevesebb üzenetre van szükségünk.

Az elkészült lekérdezési fából elkészítjük a végső lekérdezi tervet, amit az *addAcceptHeader* végez. A korábbi megoldásokban meg volt kötve, hogy egy iterációban csak egy címkével rendelkező élen mehet csak adat. Emiatt annyi iterációra van szüksé-



5.10. ábra. Párhuzamos lekérdezési terv példa. A bal felső sarokban található hármasminták alapján készített kiértékelési fa, data property-k és a lekérdezési terv.

günk, ahány Object property szerepel a lekérdezésben. Ez lényegében lineáris kiértékelést jelent a lekérdezési tervben, de ahogy láthatjuk a lekérdezési fán: vannak olyan élek, amelyeket lehet párhuzamosan futtatni egy iteráción belül. Az *AddAcceptHeaders* függvényt kiegészítjük azzal, hogy meghatározza a hármasminták iteráció sorszámát, ami mindig a forrás sorszáma+1. Láthatjuk, hogy a kiértékelési szintek a különböző hármasmintáknál azonosak lehetnek, vagyis azokat a feltételeket azonos iterációban is kiértékelhetjük. Ezzel azt érjük el, hogy a szükséges iterációk számát lecsökkentjük a kiértékelési fa mélységére. Az üzenetek száma ezzel a megoldással nem változik, hisz a küldési feltételek ugyanazok, mint a Sparkql megvalósításnál.

A másik előnye ennek a megoldásnak, hogy nincs szükségünk *alive* üzenetekre. A probléma, hogy ha a fa ágait rossz sorrendbe értékeljük ki, akkor eljuthatunk egy olyan állapotba, hogy valamely él soha nem lesz kiértékelve, hisz nem az előző iterációban kapta az üzenetet. A párhuzamos kiértékeléssel viszont elértük azt, hogy egy iterációban a fa egy szintjét értékeltük ki, így a következő szinten lévő élek aktívak maradtak. Ennek köszönhetően minden él aktív volt, amikor a kiértékelése történt. Ezzel a megoldással elhagyhatjuk az éleket életben tartó *alive* üzeneteket és csökkenthetjük a szükséges üzenetek számát.

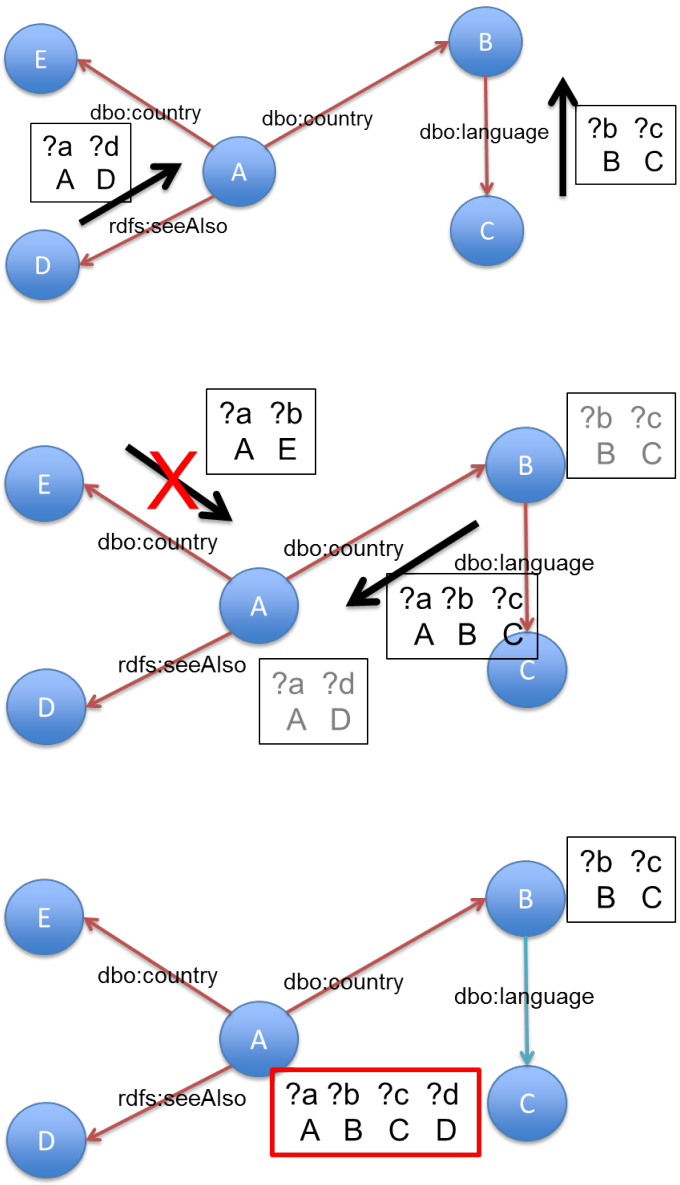
A 5.11. ábra egy példán mutatja be az algoritmus működését. A lekérdezésünkben szereplő hármasminták: *?a rdf:seeAlso ?d* , *?a dbo:country ?b* , *?b dbo:language ?c* . Ebben az esetben a lekérdezési fa gyökér változója a *?a* lesz. Az algoritmus az első iterációban

Algoritmus 25 Párhuzamos lekérdezési terv készítés

```
1: function createPlan(Query, dataProps, objProps)
2:     (objPropPatterns, dataPropPatterns) = splitQuery(Query, dataProps)
3:     objPropPatterns = orderByMaxProp(objPropPatterns, objProps)
4:     rootNode = "", vars = {},
5:     q = Queue.init()
6:     plan = List(PlanItem).init()
7:     if objPropPatterns.size > 0 then
8:         rootNode = objPropPatterns.first().s
9:         q.push(rootNode)
10:    end if
11:    while q is not empty  do
12:        actVar = q.first()
13:        for all pattern ∈ getNotProcessedPatterns(actVar, objPropPatterns) do
14:            vars.add(pattern.s)
15:            vars.add(pattern.o)
16:            if pattern.s <> actVar then
17:                q.pushIfNotContains(pattern.o)
18:                plan.append(PlanItem(pattern, pattern.o))
19:            else
20:                q.pushIfNotContains(pattern.s)
21:                plan.append(PlanItem(pattern, pattern.s))
22:            end if
23:            setPatternToProcessed(pattern)
24:        end for
25:    end while
26:    addAcceptHeaders(plan)
27:    return (plan, rootNode, vars, dataPropPatterns)
28: end function
```

Algoritmus 26 Elfogadható fejlécek készítése iterációszámmal

```
1: function addAcceptHeaders(plan)
2:     headers.init()
3:     levels.init()
4:     plan.reverse
5:     for all planItem ∈ plan do
6:         src = planItem.src
7:         des = getOther(planItem.pattern, src)
8:         headers(des).addAll(headers(src))
9:         headers(des).add(src)
10:        levels(des).add(levels(src) + 1)
11:        planItem.setAcceptHeaders(headers(src))
12:        planItem.setLevels(levels(src))
13:    end for
14: end function
```



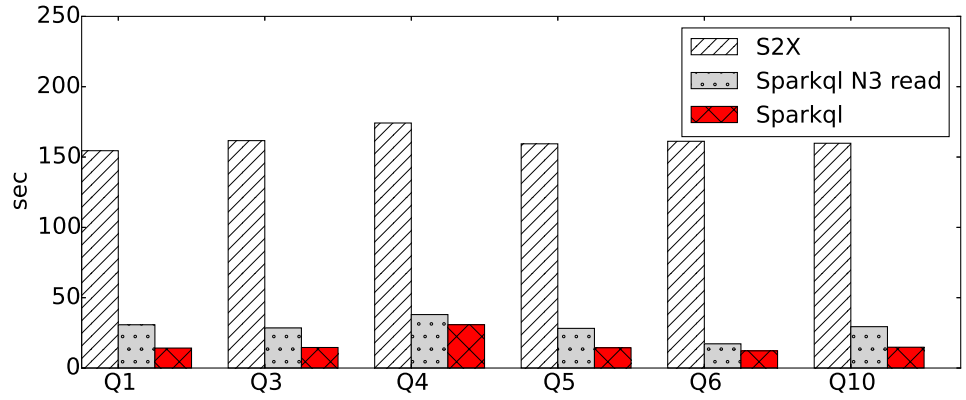
5.11. ábra. Példa a P-Sparkql működésére. 1- a megfelelő éleken ellenőrizzük hogy vannake korábbi részeredmények. 2- Ha megfelelőek a rész eredmények, akkor elküldjük a megfelelő irányba az üzenetet. 3- A cél csomópont összekapcsolja az eredményt a saját részeredményével. 4- Ezután lépünk a következő iterációba, ahol kezdjük előröl a folyamatot. 5- Végül csak a gyökér változókkal rendelkező csomópontoknál lesz megtalálható az eredmény.

(bal kép) az *rdfs:seealso* és a *dbo:language* állítmányokat alkalmazza. Ebben az esetben nincsenek egyéb feltételeink, így a C és D csomópontok elküldik az éllel készített részeredményeiket. A következő iterációban (középső kép) a *dbo:country* állítmány alkalmazzuk. Láthatjuk hogy az E és a B csomópont rendelkezik megfelelő éllel a kiértékeléshez. Azonban csak a B csomópont részeredménye érvényes, mert ő kapott információt az előző iterációban a *?c* változóról. A B a részeredményéhez kapcsolja hozzá a *dbo:country* élt, majd küldi el az eredményt az A csomópontnak. A végső eredmény (jobb kép) az A csomópontnál lesz tárolódva. Az eredmények összegyűjtése úgy történik, hogy vesszük azokat a csúcsokat, amelyek a gyökér elem változót helyettesítik be. Ezek közül csak azok lesznek eredmények, amelyek tartalmazzák az összes változót. A példánkban a *?a* gyökér elem és az A tartalmaz az összes változóhoz értéket.

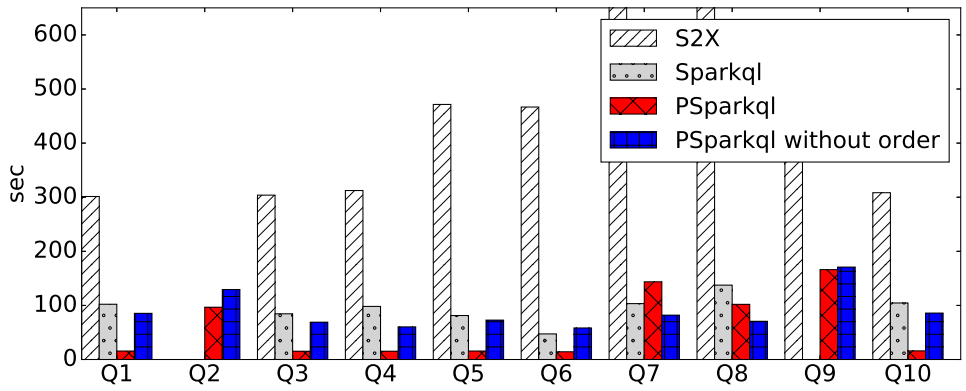
5.6. Mérési eredmények

Az eredményeimet összevetettem az eddig elérhető megoldással, ami a Schätzl és társai által készített S2X. Az elemzéshez a LUBM adat generátort és a teszt lekérdezéseket alkalmaztam. A teszteléshez 20 egyetemet generáltam, majd egy egyszerű következtetővel elkészítettem azokat a hármasokat, amelyek szükségesek a LUBM lekérdezések megválaszolásához. Az egyik ilyen például, hogy minden *PostGraduateStudent Student* is. A következtetések elkészítése után 8,5 millió hármasom lett az elemzések elvégzéséhez. A teszt egy VmWare virtualizációval létrehozott virtuális gépen futott. A szerver 24 maggal (2,5 GHz) és 128 GB Ram memóriával rendelkezett. A virtuális gép megkapta az összes magot és 64GB RAM memóriát. A teszteléshez a Spark 2.0.1-es verzióját alkalmaztam. A mérésekhez minden lépésben leállítottam és újraindítottam a Worker node-ot, így elkerülve az esetleges gyorsítótárból adódó eredményeket. A Spark master és a Worker is ezen a virtuális gépen lett elindítva. A Worker node-nak 40 GB RAM-ot és 20 magot adtam. A maradékot a master használta. A Sparkot natívan alkalmaztam, nem használtam klaszter management rendszereket (pl. Yarn) és osztott filerendszert sem (HDFS). Az input fileok a lokális filerendszerről lettek beolvasva.

A 5.12 ábrán láthatjuk a futási időket a Sparkql rendszernek és az S2X-nek. Láthatjuk, hogy a Sparkql gyorsabban tudott lefutni, mint az S2X. Összevetettem a futási időket az N3 formátumú olvasással és az átkódolt formátummal is. Láthatjuk, hogy az átkódolt formátummal a futási idő kevesebb, de nem annyival, hogy megérje ezt az új formátumot elkészíteni. A szemantikus adatok általában elérhetőek N3 formátumban, így a kódolt formátumot nem használtam a későbbiekben.



5.12. ábra. Lekérdezési idők a LUBM lekérdezésekre S2X és Sparkql külöböző beolvasási módszerekkel



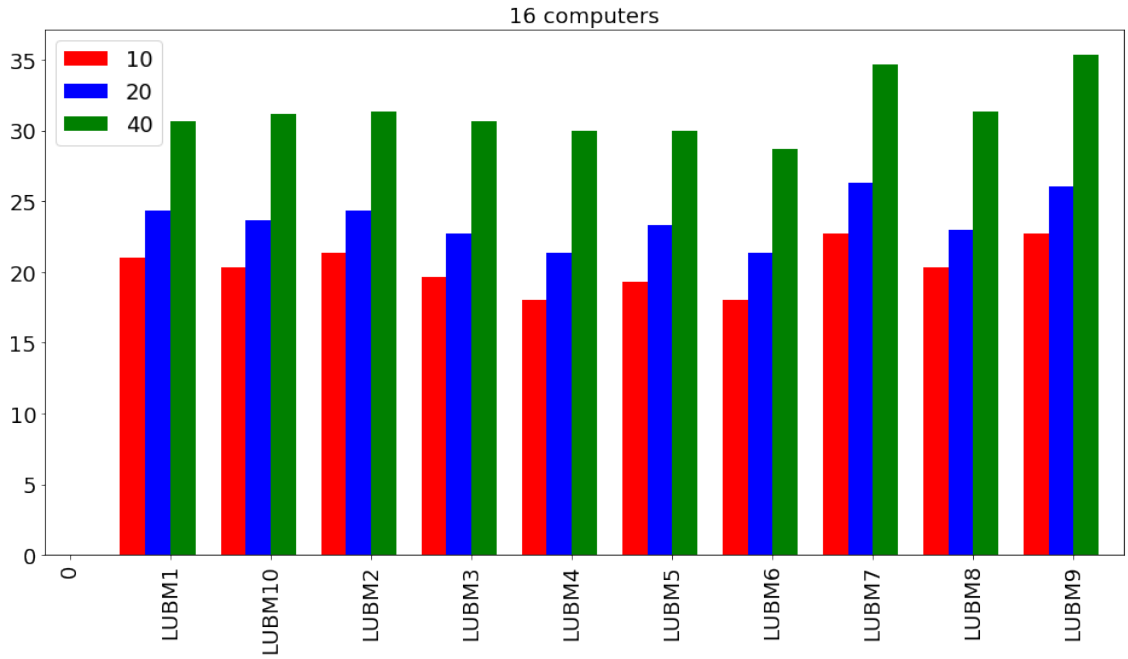
5.13. ábra. Lekérdezési idők a LUBM lekérdezésekre S2X, Sparkql és PSparkql statisztikával és statisztika nélkül

5.2. táblázat. Méréshez használt adathalmazok méretei

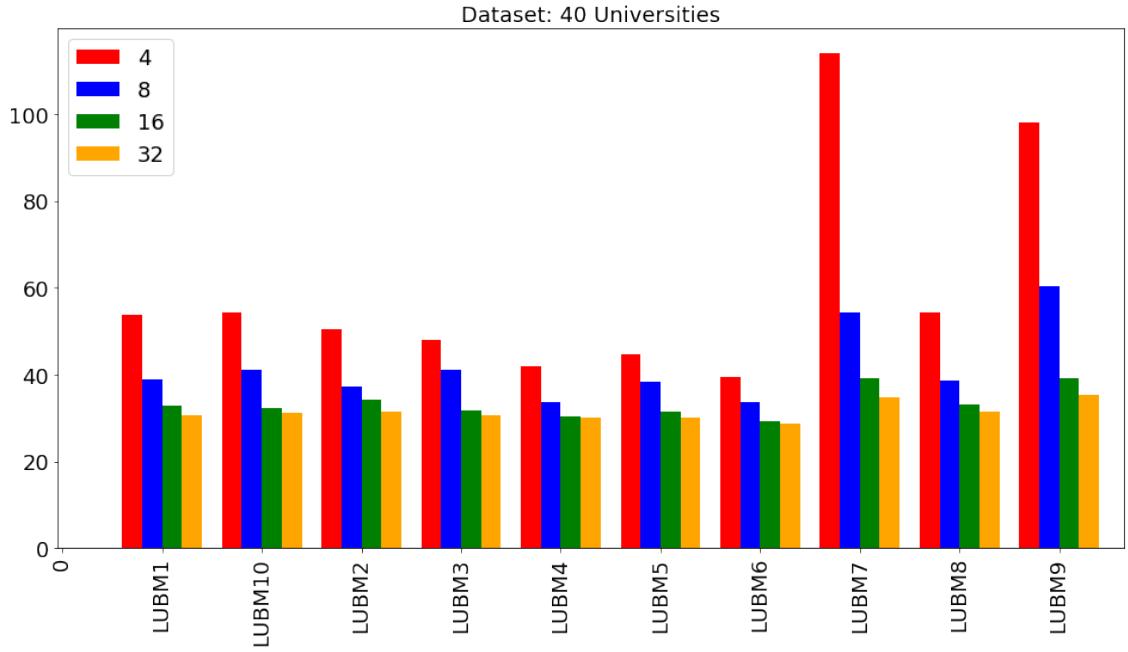
Egyetemek száma	Hármasok szám	Adatméret
10	1 667 968	239 MB
20	3 250 344	467 MB
40	6 340 817	910 MB

A 5.13 ábrán láthatjuk a futási időket különböző rendszereken. Láthatjuk, hogy az S2X-nek van szüksége a legtöbb időre, hogy ki tudja értékelni a SPARQL lekérdezést. Az idő felét a szemantikus gráf beolvasására használja, a másik fele szükséges a kiértékelésre. A Sparkql gyorsabban tudja kiértékelni a lekérdezést, de sok felesleges üzenet és él szükséges a gráf létrehozásához. A grafikonon láthatjuk azt is, hogy vannak olyan lekérdezések amelyek nem adtak eredményt. Az S2X és a Sparkql is memória hibával nem tudott lefutni a Q2 lekérdezésre. A Q7 és Q8 lekérdezésekre a S2X-szel kaptunk választ, de a kiértékeléshez 7 órára volt szüksége. A Sparkql viszont a Q9-esre nem tudott választ adni, és memória hibával leállt. Láthatjuk, hogy a PSparkql viszont minden lekérdezésre tudott választ adni. Ennek oka, hogy a csökkentett él és üzenetszám miatt kevesebb memóriára volt szüksége. A PSparkql rendszert megvizsgáltam úgy is, hogy használja a statisztika alapú lekérdezést. Ennek lényege, hogy olyan él, amely ritkán szerepel, az fusson az első iterációban, a további iterációk pedig az elfogadott részeredmények miatt csökkenteni tudják a szükséges élek számát. Láthatjuk, hogy ha használjuk ezt a megoldást, akkor a lekérdezések általában gyorsabbak lettek, mint nélküle. Viszont vannak olyan lekérdezések, amelyeknél ez rontott az eredményen. Láthatjuk, hogy a Q7 és Q8 lekérdezések lassabbak lettek. Ennek oka, hogy az eredeti lekérdezésben olyan hármasok vannak a lekérdezés első sorában, amelyek konstans értéket tartalmaznak. A statisztika alapja viszont a property-k számossága, így ez a hármas a statisztika alapján később fut le. Az eredeti lekérdezés pedig gyorsabban tudja szűrni azt a csúcsot, amelyik rendelkezik ezzel az URI-val.

A 5.14 ábra szemlélteti a különböző méretű adathalmazokon a PSparkql futási idejét. Ennél a vizsgálatnál 16 számítógépet alkalmaztunk *worker node*-nak és egy *master node*-unk volt. A gépek Intel Core i7-6700 3,4GHz-es processzorral és 8 GB memóriával rendelkeztek. Az adathalmazokat a LUBM adathalmaz generátorral készítettem. A méreteket a 5.2. táblázatban láthatjuk. Láthatjuk, hogy a méretek növelésével nem változtak jelentősen a futási idők. A 5.15. ábra pedig azt mutatja meg, hogy hogyan változik a futási idő, ha változtatjuk a klaszterben található számítógépek számát. Ahogy várható volt a futási idők javultak ha több csomópont volt képes számolni. A leglényegesebb különbséget a LUBM7 lekérdezésnél láthatunk, ahol a futási idő több mint a felére csökkent.



5.14. ábra. Lekérdezési idők a LUBM lekérdezésekre 10-20-40 egyetemet tartalmazó adathalmazra 16 gépen.



5.15. ábra. Lekérdezési idők a LUBM lekérdezésekre 40 egyetemet tartalmazó adathalmazra 4-8-16-32 gépet használva.

Ennek a lekérdezésnek a lényege, hogy tesztelje a rendszert olyan lekérdezéssel, amelynek az eredménye nagy méretű. Itt a párhuzamos feldolgozás előnye tud érvényesülni.

5.7. Összefoglalás és kapcsolat a tézisekkel

A szemantikus adatok nagyban hasonlítanak a Big Data adatokra, amelyek a méretük vagy a strukturálatlanságuk miatt nehezen kezelhetőek a mai adatbázis rendszerekkel. A szemantikus adatok kezelése szintén ezen problémák miatt nehézkes. Ebben a fejezetben bemutattam olyan algoritmusokat, amelyek a Big Data eszközök segítségével képesek a szemantikus adatok kezelésére. Az első algoritmus arra szolgált, hogy redukálni tudjuk a szemantikus gráfot. A módszer lényege, hogy a hasonló csúcsokat összevonva, és csak bizonyos éleket megtartva a gráfot le tudtuk csökkenteni olyan méretűre, amelyet már egy egyszerű gráfmegjelenítő alkalmazás is képes kezelni. Ez a megoldás a 5. tézisem, azaz a MapReduce módszer alkalmazható szemantikus gráfok redukálására. Ez rámutatott az adathalmazban rejlő hibákra is. A másik algoritmus a szemantikus adatok lekérdezésére szolgált a Spark GraphX rendszeren. Ez a rendszer egy osztott gráf elemző eszköz, amely a Bulk Synchronous Parallel (BSP) modellen alapuló Pregel [76] modellt alkalmazza. A modell lényege, hogy a gráfban lévő csúcsok számolnak értékeket a szomszédoktól kapott üzenetek alapján. Erre a modellre készítettem el a Sparkql alkalmazást. A megoldásban szükség volt egy lekérdezési terv megvalósítására, a csúcsok és az üzenetek kezelésére. Ebben a megoldásban a lekérdezési terv lineárisan értékelte ki a SPARQL lekérdezést, amely a 6. tézisem. A 7. tézisem pedig kimondja, hogy lekérdezési tervet lehet párhuzamosítani újfajta üzenet modellel. Ezzel a megoldással nincs szükség hurok élekre és 'alive' üzenetekre a gráfban. Ezt a javítást hívtam PSparkql algoritmusnak.

6. fejezet

Összegzés

Az értekezés a szemantikus webnél felmerülő problémákkal foglalkozott. A szemantikus web egy olyan elgondolás, ahol az Interneten információkat tárolunk és ezek összekapcsolásából egy nagy tudásbázist képezünk. Ezzel viszont több probléma is felmerül. A problémák fő oka, hogy ezek a tudásbázisok nagy méretűek és strukturálatlanok, ami miatt nem tudjuk őket egyszerűen lekérdezni. Az információkat egyszerű állításokként írjuk le alany-állítmány-tárgy formában. Az itt található alany és tárgy egy azonosító, amelyek más állításokban is szerepelhetnek. Emiatt az adatokat felfoghatjuk úgy is, mint egy nagy gráf, ahol az alany és állítmány a csúcsok, a tárgy pedig a köztük levő címkézett él. Emiatt nevezhetjük a szemantikus adatokat strukturálatlan adatoknak.

A 3. fejezet azt mutatja be, hogyan lehet használni a kliens-szerver architektúra modellt arra, hogy a szemantikus adatokat biztosítani tudjunk kisebb erőforrással rendelkező eszközöknek, mint például a mobiltelefonoknak. A fejezet bemutat három alkalmazást, amely ezen architektúra felépítésre épül. Az első alkalmazás egy vállalati információs rendszer, amelyben az adatokat és a jogokat szemantikus formátumban tároljuk. A második egy beltéri navigációs rendszer, mely a szemantikus információk alapján készíti el az útvonalat épületen belül. A harmadik pedig egy szemantikus böngésző mobilkészülékekre, amelyeknek a célja, hogy képesek legyünk a szemantikus adatok megismerésére. A legtöbb ilyen alkalmazás nem elérhető mobil készülékek számára, de a fejezetben bemutatott architektúra lehetővé tette ezek használatát.

A 4. fejezet a federált rendszerek témakörével foglalkozik. A szemantikus web elgondolása, hogy a világhálón megtalálható információk összekapcsolhatóak legyenek. A SPARQL 1.1 óta a SERVICE kulcsszó segítségével valósítható meg olyan lekérdezés, amely több forrást, úgynevezett SPARQL végpontot képes alkalmazni. A legtöbb végpont viszont nem támogatja ezeket a lekérdezéseket, hisz az ilyen lekérdezések költsége nagy. A végpontnak le kell kérdeznie a többi végpontot és a kapott eredményeket össze kell kapcsol-

nia. Másfelől egy ilyen lekérdezés megírása se egyértelmű, hisz ahhoz, hogy össze tudjunk kapcsolni két adathalmazt ismernünk kell az elérésüket, illetve az ott tárolt adatokat. A federált rendszerek erre adnak megoldást, hisz ezeknek egy SERVICE nélküli lekérdezést kell megadnunk, amelyből a rendszer dönti el, hogy az egyes hármasmintákat mely végpontokon kell lefuttatni. A fejezet első eredménye egy olyan megoldást mutatott be ahol a federált rendszer a végpontokon található névtereket használja a kiválasztásra. A második eredmény arra irányul, hogy egy federált rendszert alkalmazni lehet szemantikus lekérdezések megírására is, amely segítheti a szemantikus weben nem járatos felhasználókat. A megoldás hármasmintákat ajánl a felhasználónak a különböző végpontokról. A kiválasztott hármasminta alapján újabb ajánlásokat tudunk készíteni, addig amíg el nem készül a végső lekérdezés. A fejezet végső eredménye, hogy ezekből az ajánlásokból olyan információkat is ki tudunk nyerni, amely segíti a lekérdezés kiértékelését. Ezek az információk lehetnek a fejezet elején említett névterek, vagy az állítmányok.

Az 5. fejezet a szemantikus adatok és a Big Data eszközök kapcsolatát írja le. Ahogy említettem a szemantikus adatok mérete nagy, emiatt merül fel a Big Data eszközök használata. A fejezet első eredményeként egy olyan algoritmust készítettem, amely a szemantikus gráfot redukálja olyan méretűre, amelyet már meg tudnak jeleníteni a gráfmegjelenítő alkalmazások. A vizuális megjelenítés segít az adatok megértésében, megismerésében. A fejezetben bemutattam pár példát arra is, hogy a módszer segítségével találtam hibásan felvitt adatokat az adathalmazban. A következő eredmény egy olyan algoritmus és modell, ahol egy osztott gráfelemző alkalmazást alkalmaztam a szemantikus adatok lekérdezésére. Ez a rendszer a Spark GraphX, amely a csúcsok közötti üzenetváltáson alapszik. A fejezetben bemutattam két megoldást is, ahol a Sparkql lekérdezési terve lineárisan megy végig a hármasmintákon, még a P-Sparkql már párhuzamosan képes feltételeket ellenőrizni.
Összegezve a következő eredmények születtek:

- Egy szemantikus adatokhoz való hozzáférést biztosító szerver alkalmazás lehetőséget ad a vékony klienseknek (mobileszközöknek) a komplex és federált lekérdezések futtatására. [2, 3, 6] (Tézis 1.)

- A federált rendszerek segítségével ajánlhatunk hármasmintákat, amely segít a felhasználónak SPARQL lekérdezés elkészítésében. Ennek formális leírása ASM modell-lel valósítható meg. [5] (Tézis 2.)

- Az ajánlásokból kinyerhetőek olyan információk, amelyek támogatják a federált rendszerek végpontválasztását. [8] Ilyen információ lehet például a végpontokon ta-

lálható névterek is. [4] (Tézis 3.)

- Biszimulációval le lehet csökkenteni a szemantikus gráfokat olyan méretűre hogy azok megjeleníthetőek legyenek gráfmegjelenítő alkalmazásokkal. A biszimulációs algoritmusok megvalósíthatóak MapReduce segítségével, ami a párhuzamos feldolgozást teszi lehetővé. [7] (Tézis 4.)

- Osztott gráfelemző rendszerek alkalmasak szemantikus lekérdezések kiértékelésére lineáris és párhuzamos lekérdezési tervvel. [9, 10] (Tézis 5., Tézis 6.)

A. függelék

LUBM lekérdezések

```
Source: http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

# Query1
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse
http://www.Department0.University0.edu/GraduateCourse0}

# Query2
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:memberOf ?Z .
  ?Z ub:subOrganizationOf ?Y .
  ?X ub:undergraduateDegreeFrom ?Y}
```

```
# Query3
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Publication .
  ?X ub:publicationAuthor
        http://www.Department0.University0.edu/AssistantProfessor0}

# Query4
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE
{?X rdf:type ub:Professor .
  ?X ub:worksFor <http://www.Department0.University0.edu> .
  ?X ub:name ?Y1 .
  ?X ub:emailAddress ?Y2 .
  ?X ub:telephone ?Y3}

# Query5
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Person .
  ?X ub:memberOf <http://www.Department0.University0.edu>}

# Query6
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X WHERE {?X rdf:type ub:Student}
```

```
# Query7
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Course .
  ?X ub:takesCourse ?Y .
  <http://www.Department0.University0.edu/AssociateProfessor0>,
   ub:teacherOf, ?Y}

# Query8
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Department .
  ?X ub:memberOf ?Y .
  ?Y ub:subOrganizationOf <http://www.University0.edu> .
  ?X ub:emailAddress ?Z}

# Query9
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
{?X rdf:type ub:Student .
  ?Y rdf:type ub:Faculty .
  ?Z rdf:type ub:Course .
  ?X ub:advisor ?Y .
  ?Y ub:teacherOf ?Z .
  ?X ub:takesCourse ?Z}
```

```
# Query10
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
{?X rdf:type ub:Student .
  ?X ub:takesCourse
<http://www.Department0.University0.edu/GraduateCourse0>}
```

Szerző publikációi

[1] **Gergő Gombos**. "SPARQL lekérdezést futtató Webservice mobil alkalmazások számára". MA thesis. ELTE IK, 2012.

[2] **Gergő Gombos** et al. "A Mobile browser prototype for semantic infromation systems". In: *Acta Electrotechnica et Informatica* 13.4 (2013), pp. 20–25.

[3] Tamás Matuszka, **Gergő Gombos**, and Attila Kiss. "A New Approach for Indoor Navigation Using Semantic Webtechnologies and Augmented Reality". In: *Virtual Augmented and Mixed Reality. Designing and Developing Augmented and Virtual Environments*. Springer, 2013, pp. 202–210.

[4] **Gergő Gombos** and Attila Kiss. "SPARQL Processing over the Linked Open Data with Automatic Endpoint Detection". In: *Advanced Approaches to Intelligent Information and Database Systems*. Springer, 2014, pp. 183–192.

[5] **Gergő Gombos** and Attila Kiss. "SPARQL Query Writing with Recommendations Based on Datasets". In: *Human Interface and the Management of Information. Information and Knowledge Design and Evaluation*. Springer International Publishing, 2014, pp. 310–319.

[6] Tamás Matuszka, **Gergő Gombos**, and Attila Kiss. "mswb: Towards a mobile semantic web browser". In: *Mobile Web Information Systems*. Springer, 2014, pp. 165–175.

[7] Gábor Rácz, **Gergő Gombos**, and Attila Kiss. "Visualization of Semantic Data Based on Selected Predicates". In: *Transactions on Computational Collective Intelligence XIV*. Springer, 2014, pp. 180–195.

[8] **Gergő Gombos** and Attila Kiss. "Federated Query Evaluation Supported by SPARQL Recommendation". In: *International Conference on Human Interface and the Management of Information*. Springer. 2016, pp. 263–274.

[9] **Gombos, Gergő**, Gábor Rácz, and Attila Kiss. "Spar (k) ql: SPARQL evaluation method on Spark GraphX". In: *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on*. IEEE. 2016, pp. 188–193.

[10] **Gombos, Gergő** and Attila Kiss. "P-Spar(k)ql: SPARQL Evaluation Method on Spark GraphX with Parallel Query Plan". In: *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE. 2017, pp. 212–219.

Szerző további publikációi

[11] Antal Iványi et al. “Parallel enumeration of degree sequences of simple graphs II”. In: *Acta Universitatis Sapientiae, Informatica* 5.2 (2013), pp. 245–270.

[12] Antal Iványi et al. “Score sets in multitournaments I. Mathematical results”. In: *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE SECTIO COMPUTATORICA*. Vol. 40. ELTE. 2013, pp. 307–320.

[13] Imre Szucs, **Gergő Gombos**, and Attila Kiss. “Five Ws, one H and many tweets”. In: *Cognitive Infocommunications (CogInfoCom), 2013 IEEE 4th International Conference on*. IEEE. 2013, pp. 441–446.

[14] **Gergő Gombos** et al. “VOSD: A General-Purpose Virtual Observatory over Semantic Databases.” In: *Acta Cybern.* 21.3 (2014), pp. 353–366.

[15] **Gergő Gombos**, Attila Kiss, and Zoltán Zvara. “Performance Analysis of a Cluster Management System with Stress Cases”. In: *Acta Polytechnica Hungarica* 13.2 (2016).

[16] Sándor Laki et al. “Take your own share of the PIE”. In: *Applied Networking Research Workshop (ANRW)*. 2017.

További publikációk

[17] David Wells et al. "Guide to GPS positioning". In: *Canadian GPS Assoc.* Citeseer. 1987.

[18] S Skiena. "Dijkstra's algorithm". In: *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley* (1990), pp. 225–227.

[19] Leslie G Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[20] Roy Want et al. "The active badge location system". In: *ACM Transactions on Information Systems (TOIS)* 10.1 (1992), pp. 91–102.

[21] Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. "Computing simulations on finite and infinite graphs". In: *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on.* IEEE. 1995, pp. 453–462.

[22] Thomas Cheatham et al. "Bulk synchronous parallel computing—a paradigm for transportable software". In: *Tools and Environments for Parallel and Distributed Systems.* Springer, 1996, pp. 61–76.

[23] Tim Berners-Lee. *Notation 3–ideas about web architecture.* 1998.

[24] M David Hanson. "The Client/Server Architecture". In: *Server Management* (2000), p. 3.

[25] Mike Addlesee et al. "Implementing a sentient computing system". In: *Computer* 34.8 (2001), pp. 50–56.

[26] Tim Berners-Lee, James Hendler, Ora Lassila, et al. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 28–37.

[27] Natalya F Noy, Deborah L McGuinness, et al. *Ontology development 101: A guide to creating your first ontology.* 2001.

[28] Jörg Baus, Antonio Krüger, and Wolfgang Wahlster. "A resource-adaptive mobile navigation system". In: *Proceedings of the 7th international conference on Intelligent user interfaces*. ACM. 2002, pp. 15–22.

[29] Zsolt N Németh and Vaidy Sunderam. "A formal framework for defining grid systems". In: *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*. IEEE. 2002, pp. 202–202.

[30] Martin Dürst and Michel Suignard. *Internationalized resource identifiers (IRIs)*. Tech. rep. 2004.

[31] Ian Horrocks et al. "SWRL: A semantic web rule language combining OWL and RuleML". In: *W3C Member submission* 21 (2004), p. 79.

[32] Brian McBride. "The resource description framework (RDF) and its vocabulary description language RDFS". In: *Handbook on ontologies*. Springer, 2004, pp. 51–65.

[33] Christos Anagnostopoulos, Vassileios Tsetsos, Panayotis Kikiras, et al. "OntoNav: A semantic indoor navigation system". In: *1st Workshop on Semantics in Mobile Environments (SME05), Ayia*. Citeseer. 2005.

[34] Ian Horrocks et al. "Semantic web architecture: Stack or two towers?" In: *International Workshop on Principles and Practice of Semantic Web Reasoning*. Springer. 2005, pp. 37–41.

[35] Michael Kifer et al. "A realistic architecture for the semantic web". In: *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer. 2005, pp. 17–29.

[36] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. "Uniform resource identifier (URI): Generic syntax". In: (2005).

[37] Veljo Otsason et al. "Accurate GSM indoor localization". In: *International conference on ubiquitous computing*. Springer. 2005, pp. 141–158.

[38] Peter F Patel-Schneider. "A revised architecture for semantic web reasoning". In: *International Workshop on Principles and Practice of Semantic Web Reasoning*. Springer. 2005, pp. 32–36.

[39] Max L Wilson et al. "mspace mobile: A mobile application for the semantic web". In: (2005).

[40] Tim Berners-Lee et al. "Tabulator: Exploring and analyzing linked data on the semantic web". In: *Proceedings of the 3rd international semantic web user interaction workshop*. Vol. 2006. Citeseer. 2006, p. 159.

[41] Graham Klyne and Jeremy J Carroll. "Resource description framework (RDF): Concepts and abstract syntax". In: (2006).

[42] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *International semantic web conference*. Vol. 4273. Springer. 2006, pp. 30–43.

[43] Emmanuel Pietriga et al. "Fresnel: A browser-independent presentation vocabulary for RDF". In: *International Semantic Web Conference*. Springer. 2006, pp. 158–171.

[44] Eric Prud, Andy Seaborne, et al. "SPARQL query language for RDF". In: (2006).

[45] Bastian Quilitz. *DARQ–Federated Queries with SPARQL*. 2006.

[46] Sören Auer et al. "Dbpedia: A nucleus for a web of open data". In: *The semantic web*. Springer, 2007, pp. 722–735.

[47] Christian Bizer and Tobias Gauß. *Disco-Hyperdata Browser: A simple browser for navigating the Semantic Web*. 2007.

[48] Jorge Cardoso, Martin Hepp, and Miltiadis D Lytras. *The semantic web: real-world applications from industry*. Vol. 6. Springer Science & Business Media, 2007.

[49] Leonidas Deligiannidis, Krys J Kochut, and Amit P Sheth. "RDF data exploration and visualization". In: *Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*. ACM. 2007, pp. 39–46.

[50] Uwe Grossmann, Markus Schauch, and Syuzanna Hakobyan. "RSSI based WLAN indoor positioning with personal digital assistants". In: *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2007. IDAACS 2007. 4th IEEE Workshop on*. IEEE. 2007, pp. 653–656.

[51] Harlan Hile and Gaetano Borriello. "Information overlay for camera phones in indoor environments". In: *International Symposium on Location-and Context-Awareness*. Springer. 2007, pp. 68–84.

[52] Apache Jena. *semantic web framework for Java*. 2007.

[53] Davide Merico and Roberto Bisiani. "Indoor navigation with minimal infrastructure". In: *Positioning, Navigation and Communication, 2007. WPNC'07. 4th Workshop on*. IEEE. 2007, pp. 141–144.

[54] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. "Yago: a core of semantic knowledge". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 697–706.

[55] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[56] Christian Becker and Christian Bizer. "DBpedia Mobile: A Location-Enabled Linked Data Browser." In: *Ldow* 369 (2008), p. 2008.

[57] Kurt Bollacker et al. "Freebase: a collaboratively created graph database for structuring human knowledge". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1247–1250.

[58] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[59] Jiří Dokulil and Jana Katreniaková. "Visual exploration of rdf data". In: *SOFSEM 2008: Theory and Practice of Computer Science*. Springer, 2008, pp. 672–683.

[60] Aurona Gerber, Alta Van der Merwe, and Andries Barnard. "A functional semantic web architecture". In: *The Semantic Web: Research and Applications* (2008), pp. 273–287.

[61] Mordechai Haklay and Patrick Weber. "Openstreetmap: User-generated street maps". In: *IEEE Pervasive Computing* 7.4 (2008), pp. 12–18.

[62] Michael Kifer. "Rule interchange format: The framework". In: *International Conference on Web Reasoning and Rule Systems*. Springer. 2008, pp. 1–11.

[63] Tsutomu Miyashita et al. "An augmented reality museum guide". In: *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*. IEEE Computer Society. 2008, pp. 103–106.

[64] Keith Alexander and Michael Hausenblas. "Describing linked datasets-on the design and usage of void, the'vocabulary of interlinked datasets". In: *Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09)*. Citeseer. 2009.

[65] Sean Bechhofer. "OWL: Web ontology language". In: *Encyclopedia of database systems*. Springer, 2009, pp. 2008–2009.

[66] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked data-the story so far". In: *Semantic services, interoperability and web applications: emerging concepts* (2009), pp. 205–227.

[67] Orri Erling and Ivan Mikhailov. "RDF Support in the Virtuoso DBMS". In: *Networked Knowledge-Networked Media*. Springer, 2009, pp. 7–24.

[68] Oktie Hassanzadeh and Mariano P Consens. "Linked Movie Data Base." In: *LDOW*. 2009.

[69] Norman Heino et al. "Developing semantic web applications with the ontowiki framework". In: *Networked Knowledge-Networked Media*. Springer, 2009, pp. 61–77.

[70] Haosheng Huang and Georg Gartner. "A survey of mobile indoor navigation systems". In: *Cartography in Central and Eastern Europe*. Springer, 2009, pp. 305–319.

[71] Mohammad Farhan Husain et al. "Storage and retrieval of large rdf graph using hadoop and mapreduce". In: *Cloud computing*. Springer, 2009, pp. 680–686.

[72] Alessandro Mulloni et al. "Indoor positioning and navigation with camera phones". In: *IEEE Pervasive Computing* 8.2 (2009).

[73] Ulrik Brandes et al. *Graph markup language (GraphML)*. Bibliothek der Universität Konstanz, 2010.

[74] Bin Chen et al. "Chem2bio2rdf: A linked open data portal for systems chemical biology". In: *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*. Vol. 1. IEEE. 2010, pp. 232–239.

[75] M Husain et al. "Data intensive query processing for large RDF graphs using cloud computing tools". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE. 2010, pp. 1–10.

[76] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[77] Konstantin Shvachko et al. "The hadoop distributed file system". In: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee. 2010, pp. 1–10.

[78] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets." In: *HotCloud* 10 (2010), pp. 10–10.

[79] Barry Bishop et al. "Factforge: A fast track to the web of data". In: *Semantic Web* 2.2 (2011), pp. 157–166.

[80] Carlos Buil-Aranda, Marcelo Arenas, and Oscar Corcho. "Semantics and optimization of the SPARQL 1.1 federation extension". In: *Extended Semantic Web Conference*. Springer. 2011, pp. 1–15.

[81] Timofey Ermilov et al. "Ontowiki mobile–knowledge management in your pocket". In: *Extended Semantic Web Conference*. Springer. 2011, pp. 185–199.

[82] Olaf Görlitz and Steffen Staab. "Federated data management and query optimization for linked open data". In: *New Directions in Web Data Management 1*. Springer, 2011, pp. 109–137.

[83] Mohammad Farhan Husain et al. "Scalable complex query processing over large semantic web data using cloud". In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 187–194.

[84] Moritz Kessel and Martin Werner. "SMARTPOS: Accurate and precise indoor positioning on mobile phones". In: *Proceedings of the First International Conference on Mobile Services, Resources, and Users, MOBILITY*. 2011, pp. 158–163.

[85] Jens Lehmann and Lorenz Bühmann. "AutoSPARQL: Let users query your knowledge base". In: *The Semantic Web: Research and Applications*. Springer, 2011, pp. 63–79.

[86] Jó Agila Bitsch Link et al. "Footpath: Accurate map-based indoor navigation using smartphones". In: *Indoor Positioning and Indoor Navigation (IPIN), 2011 International Conference on*. IEEE. 2011, pp. 1–8.

[87] Steven Lynden et al. "Aderis: An adaptive query processor for joining federated sparql endpoints". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2011, pp. 808–817.

[88] Alessandro Mulloni, Hartmut Seichter, and Dieter Schmalstieg. "Handheld augmented reality indoor navigation with activity-based instructions". In: *Proceedings of the 13th international conference on human computer interaction with mobile devices and services*. ACM. 2011, pp. 211–220.

[89] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. "PigSPARQL: Mapping SPARQL to Pig Latin". In: *Proceedings of the International Workshop on Semantic Web Information Management*. ACM. 2011, p. 4.

[90] Michael Schmidt et al. "Fedbench: A benchmark suite for federated semantic data query processing". In: *The Semantic Web–ISWC 2011*. Springer, 2011, pp. 585–600.

[91] Andreas Schwarte et al. “FedX: Optimization techniques for federated query processing on linked data”. In: *The Semantic Web–ISWC 2011*. Springer, 2011, pp. 601–616.

[92] Sören Auer et al. “LODStats–an extensible framework for high-performance dataset analytics”. In: *International Conference on Knowledge Engineering and Knowledge Management*. Springer. 2012, pp. 353–362.

[93] Egon Börger and Robert Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Science & Business Media, 2012.

[94] Ming Li, Lars Mahnkopf, and Leif Kobbelt. “The design of a segway AR-Tactile navigation system”. In: *International Conference on Pervasive Computing*. Springer. 2012, pp. 161–178.

[95] Chang Liu et al. “Hadoopsparql: a hadoop-based engine for multiple sparql query answering”. In: *The Semantic Web: ESWC 2012 Satellite Events*. Springer, 2012, pp. 474–479.

[96] Nikolaos Papailiou et al. “H2RDF: adaptive query processing on RDF data in the cloud.” In: *Proceedings of the 21st international conference companion on World Wide Web*. ACM. 2012, pp. 397–400.

[97] Martin Voigt, Stefan Pietschmann, and Klaus Meißner. “Towards a semantics-based, end-user-centered information visualization process”. In: *Proc. of the 3rd international workshop on semantic models for adaptive interactive systems (SEMAIS 2012)*. 2012.

[98] Tom White. *Hadoop: The definitive guide*. O’Reilly Media, Inc., 2012.

[99] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[100] Paul A Zandbergen. “Comparison of WiFi positioning on two mobile devices”. In: *Journal of Location Based Services* 6.1 (2012), pp. 35–50.

[101] Steve Harris, Andy Seaborne, and Eric Prud’hommeaux. “SPARQL 1.1 query language”. In: *W3C recommendation* 21.10 (2013).

[102] Patrick Hoefler. “Linked Data Interfaces for Non-expert Users”. In: *The Semantic Web: Semantics and Big Data*. Springer, 2013, pp. 702–706.

[103] Kasjen Kramer, Renata Queiroz Dividino, and Gerd Gröner. "SPACE: SPARQL Index for Efficient Autocompletion." In: *International Semantic Web Conference (Posters & Demos)*. 2013, pp. 157–160.

[104] Yongming Luo et al. "Bisimulation reduction of big graphs on mapreduce". In: *British National Conference on Databases*. Springer. 2013, pp. 189–203.

[105] András Micsik, Zoltán Tóth, and Sándor Turbucz. "LODmilla: Shared Visualization of Linked Open Data". In: *International Conference on Theory and Practice of Digital Libraries*. Springer. 2013, pp. 89–100.

[106] Andriy Nikolov, Andreas Schwarte, and Christian Hütter. "Fedsearch: Efficiently combining structured queries and full-text search in a sparql federation". In: *International Semantic Web Conference*. Springer. 2013, pp. 427–443.

[107] Eric Prud'hommeaux et al. "Turtle–terse rdf triple language". In: *Candidate Recommendation, W3C* (2013).

[108] Nur Aini Rakhmawati et al. "Querying over Federated SPARQL Endpoints—A State of the Art Survey". In: *arXiv preprint arXiv:1306.1723* (2013).

[109] Laurens Rietveld and Rinke Hoekstra. "Yasgui: Not just another sparql client". In: *The Semantic Web: ESWC 2013 Satellite Events*. Springer, 2013, pp. 78–86.

[110] Alexander Schätzle et al. "Large-scale bisimulation of RDF graphs". In: *Proceedings of the Fifth Workshop on Semantic Web Information Management*. ACM. 2013, p. 1.

[111] Carlos Buil-Aranda, Axel Polleres, and Jürgen Umbrich. "Strategies for Executing Federated Queries in SPARQL1. 1". In: *The Semantic Web–ISWC 2014*. Springer, 2014, pp. 390–405.

[112] Stéphane Campinas. "Live SPARQL Auto-Completion". In: *ISWC 2014 Posters & Demonstrations Track*. CEUR-WS.org, 2014, pp. 477–480.

[113] Gavin Carothers and Andy Seaborne. "RDF 1.1 N-Triples: A line-based syntax for an RDF graph". In: *World Wide Web Consortium. http://www. w3. org/TR/n-triples/. Accessed* 24 (2014).

[114] Joseph E Gonzalez et al. "Graphx: Graph processing in a distributed dataflow framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 599–613.

[115] Eric L Goodman and Dirk Grunwald. "Using vertex-centric programming platforms to implement SPARQL queries on large graphs". In: *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press. 2014, pp. 25–32.

[116] András Micsik, Sándor Turbucz, and Zoltán Tóth. "Browsing and Traversing Linked Data with LODmilla." In: *ERCIM News* 2014.96 (2014).

[117] Alexander Schätzle et al. "Sempala: Interactive SPARQL query processing on hadoop". In: *The Semantic Web–ISWC 2014*. Springer, 2014, pp. 164–179.

[118] Ruben Verborgh et al. "Querying datasets on the Web with high availability". In: *The Semantic Web–ISWC 2014*. Springer, 2014, pp. 180–196.

[119] Ibrahim Abdelaziz et al. "Spartex: A vertex-centric framework for RDF data analytics". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1880–1883.

[120] Lushan Han et al. "Querying RDF data with text annotated graphs". In: *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. ACM. 2015, p. 27.

[121] Alexander Schätzle et al. "S2RDF: RDF Querying with SPARQL on Spark". In: *arXiv preprint arXiv:1512.07021* (2015).

[122] Alexander Schätzle et al. "S2X: Graph-Parallel Querying of RDF with GraphX". In: *Proc. of 1st International Workshop on Big-Graphs Online Querying*. 2015.

Összefoglaló

Az értekezés a szemantikus webnél [26] felmerülő problémákkal foglalkozott. A szemantikus web egy olyan elgondolás, ahol az Interneten az információkat úgy tároljuk, hogy azok összekapcsolhatóak legyenek. Ezáltal egy nagy tudásbázist kaphatunk. A használatuk viszont nehézkes, melynek fő okai a tudásbázisok nagy mérete és strukturálatlansága.

Az első terület azt mutatta be, hogy hogyan lehet használni a kliens-szerver architektúra modellt arra, hogy a szemantikus adatokat biztosítani tudjunk kisebb erőforrással rendelkező eszközöknek, mint például a mobiltelefonoknak. A fejezetben bemutattam 3 alkalmazást, amely ezen architektúra felépítésre épült. Az első alkalmazás egy vállalati információs rendszer, a második alkalmazás egy beltéri navigációs rendszer, a harmadik alkalmazás pedig egy szemantikus böngésző mobilkészülékekre.

A második terület a federált rendszerek [108] témakörével foglalkozik. A szemantikus web elgondolása, hogy a világhálón megtalálható információk összekapcsolhatóak legyenek. A federált rendszerek lehetőséget adnak arra, hogy egy lekérdezésben el tudjunk érni több végponton található adatot. Ennek fontos része, hogy a rendszer el tudja dönteni a hármasmintákból, hogy mely végponton kell lefuttatni őket. Első eredményként formálisan leírtam a federált rendszerek működését az ASM modell [93] segítségével. Ezután bemutattam egy olyan megoldást, ahol a federált rendszer a végpontokon található névtereket használja a végpontok kiválasztásra. A következő eredmény arra irányul, hogy egy federált rendszert alkalmazni lehet szemantikus lekérdezések megírására is, amely segítheti a szemantikus weben nem járatos felhasználókat. A megoldás hármasmintákat ajánl a felhasználónak a különböző végpontokról. Végül a fejezet végső eredménye, hogy ezekből az ajánlásokból olyan információkat is ki tudunk nyerni, amelyek segítik a lekérdezés kiértékelését.

A harmadik terület amivel foglalkoztam a szemantikus adatok és a Big Data eszközök kapcsolata. A fejezet első eredményeként egy olyan algoritmust készítettem, amely a szemantikus gráfot redukálja olyan méretűre, amelyet már meg tudunk jeleníteni egy gráfmegjelenítő alkalmazással. A vizuális megjelenítés segít az adatok megértésébe, megismerésébe, vagy épp a hibák feltárásában. A következő eredmény egy olyan algoritmus és modell, ahol egy osztott gráfelemző alkalmazást alkalmazok a szemantikus adatok lekérdezésére. Ez a rendszer a Spark GraphX [114], amely a csúcsok közötti üzenetváltáson alapszik. A fejezetben bemutatunk két megoldást is arra, hogyan lehet kiértékelni a SPARQL [42] lekérdezéseket. Az Sparkql megoldás lekérdezési terve lineárisan, még a P-Sparkql már párhuzamosan képes a hármasmintákat ellenőrizni.

Summary

The dissertation deals with issues of the semantic web. The semantic web [26] is a concept where information is stored on the Internet and the information can be connected each other. This way we can create a huge knowledge base which is hard to use. The main problems are the size and structure of the knowledge base.

The first research results show the client-server architecture model is usable to provide semantic data for devices with low on resources, such as mobile phones. In this chapter I presented three applications which were built on this architecture. The first one is an enterprise information system, the second one is an indoor navigation system and the third one is a semantic browser for mobile phones.

The second part of the dissertation deals with the federated systems [108]. The idea of the semantic web is that the information on the web can be connected. Federated queries provide a solution to query multiple endpoints in one query. Usually the Semantic Data is reachable over SPARQL Endpoints. The SPARQL [42] is a query language of the semantic web. The important problem of federated systems is the endpoint selection. This part of the federated system choose endpoints for the triple patterns. First I formally described the operation of the federated systems with the ASM model [93]. After that I presented an alternative solution for the endpoint selection. This technique uses the namespaces of the dataset for the endpoint selection. The next result is a triple pattern recommendation system based on the federated systems. This technique helps the user to write SPARQL queries easily. Finally I presented that recommended triple patterns store information about the endpoint and the idea uses these information for the endpoint selection.

The third research results presents the connection between the semantic web and the Big Data. The first result of the chapter is an algorithm which can reduce the semantic graph to a size that we can handle with a graph viewer. The visualization helps us to get to know and understand the graph or to detect errors from the graph. The next result is an algorithm and a model that works on a distributed graph analytic tool. I used this tool to queries of a big semantic graph. This tool is the Spark GraphX [114], which is based on the message sending between the nodes. In that chapter I presented two solutions for evaluating SPARQL. Sparkql uses a linear evaluation strategy while P-Sparkql uses a parallel evaluation strategy.

[1]ADATLAP
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai

A szerző neve: Gombos Gergő .....
MTMT-azonosító:... 10034160.....
A doktori értekezés címe és alcíme:
... Szemantikus adatok lekérdezése federált és osztott rendszereken.....
DOI-azonosító[2]: 10.15476/ELTE.2018.033
A doktori iskola neve:... Informatikai Doktori Iskola.....
A doktori iskolán belüli doktori program neve:... Információs Rendszerek.....
A témavezető neve és tudományos fokozata:..... Dr. Kiss Attila, PhD, CSc.....
A témavezető munkahelye:...ELTE Informatikai Kar, Információs Rendszerek Tanszék.....

II. Nyilatkozatok

1. A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom a.....Doktori Iskola hivatalának ügyintézőjét ....., hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: Budapest, 2018-02-28

a doktori értekezés szerzőjének aláírása

---

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.

[2] A kari hivatal ügyintézője tölti ki.

[3] A megfelelő szöveg aláhúzandó.

[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.

[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.

[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.